MASTER EN SCIENCES INFORMATIQUES

Concrétisation de tests abstraits avec AbsCon, un AddOn Qtaste

Vanhecke, Jérémy

Award date:
2016

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2015–2016

Concrétisation de tests abstraits avec AbsCon, un *AddOn* QTaste

Jeremy Vanhecke

Maître de stage : /

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Patrick Heymans

Co-promoteur : Xavier Devroey & Gilles Perrouin

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Remerciements

Je souhaiterais tout d'abord remercier les enseignants de l'Université de Namur que j'ai eu la chance de côtoyer. Vous avez tous réussi à me transmettre un peu de votre passion tout au long de ce master.

Un grand merci aussi à Patrick Heymans, Xavier Devroey et Gilles Perrouin. Vous vous êtes montrés énormément disponibles ; tant pour le démarrage du mémoire que pour les validations du contenu.

Merci à Laurent Vanboquestal qui m'a initié à l'utilisation de QTaste et qui était disponible pour répondre à mes questions.

Et enfin, merci à celles et ceux qui auront lu et apporté leurs commentaires sur ce travail.

Table des matières

Chapitre 1

Introduction

Dans la production de logiciels, la vérification complète d'un produit est souvent impossible ; par contre, en le testant, on peut montrer l'absence de certains bugs. Tester le système du point de vue utilisateur est une des méthodes existantes pour réaliser cette tâche. Pour faire une telle chose, l'idée la plus intuitive est d'exécuter des tests fonctionnels au travers de l'interface utilisateur. Elle est appelée GUI (*Graphical User Interface*) ou encore IHM (Interface Homme-Machine) pour les logiciels ayant une interaction avec l'utilisateur au travers d'un écran ; mais cette interface utilisateur peut aussi être mécanique ou électrique (par exemple : une télécommande, un distributeur, etc...).
Les désavantages principaux de cette méthode sont doubles. D'une part, le temps qu'il faut pour exécuter ces tests ; directement dépendant du nombre de tests. D'autre part, nous savons que l'interface utilisateur est l'élément d'un logiciel le plus susceptible d'être mis à jour au long de sa durée de vie. Cela signifie donc que les tests devront être adaptés après chaque changement dans l'interface.
L'exécution automatisée de tests est une réponse au problème du temps d'exécution. Cela permet de tester un système avec un minimum d'interaction humaine grâce à un programme qui va simuler celle-ci.
Le "Test Dirigé par les Modèles" (TDM ou MBT pour *Model Based Testing* en anglais) est une réponse à la problématique de variation de l'interface utilisateur. Cette méthode propose de constuire un modèle du système à tester et d'en dériver des séries de "Cas de Tests Abstraits" (CTAs) qui seront une modélisation abstraite des tests à réaliser sur le système ; ils ne sont donc pas directement exécutables en tant que tels et sont plutôt directement lisibles par un lecteur *lambda*.
A l'heure actuelle, nous pouvons trouver quelques "Chaines d'Outils de Tests TDM" qui permettent de définir le modèle du système, et qui ensuite, génèrent des cas de tests abstraits puis les convertissent en tests exécutables sur le système. La majorité de ces outils sont spécifiques à un type de système. Par exemple, Lasalle *et al* [21] proposent dans une publication un outil complet pour ce genre de tâche, mais il est imaginé pour tester des systèmes de mécanique automobile uniquement.

Dans ce travail, nous essayerons de savoir si une solution plus universelle existe. Pourrions-nous concrétiser de la même manière des CTAs destinés à des sytèmes de types différents ? Et comment pourrions-nous gérer l'exécution de ces tests concrétisés alors qu'ils visent, encore une fois, des systèmes différents ?

Dans le deuxième chapitre, nous parlerons de la problématique des tests et des méthodes de test utilisées pour mettre à l'épreuve les logiciels. Ensuite, une brève

mise au point sera faite sur le TDM qui est utilisé pour générer des Cas de Tests Abstraits. Enfin, nous analyserons certaines solutions existantes pour aider à la concrétisation et/ou l'exécution des tests.
Dans les chapitres 3 et 4, nous tenterons d'apporter une réponse satisfaisante aux questions posées dans le paragraphe précédent.
Finalement, une discussion et une conclusion regroupant tous les éléments phares de cette étude permettront d'évaluer ce travail.
La Figure 1.1 représente un résumé de la situation actuelle :

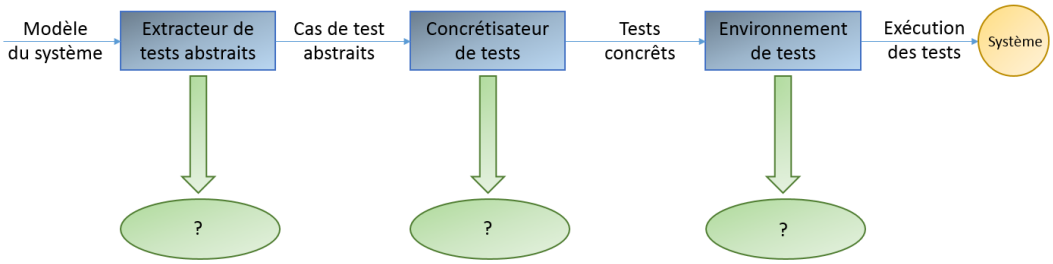


FIGURE 1.1 – L'envergure du problème

Chapitre 2

La problématique

Dans ce chapitre, nous évoquerons les méthodes utilisées dans l'industrie du logiciel pour tester leurs produits. Nous passerons en revue la notion de TDM et expliquerons pourquoi celle-ci est une bonne réponse aux tests fonctionnels. Ensuite, nous analyserons les outils existants qui pourraient nous aider pour la concrétisation et/ou l'exécution de tests. Finalement, nous ferons le point sur ces outils et analyserons leurs limites.

2.1 Approches de test

Fournir les preuves qu'un produit logiciel ne contient aucun bug se trouve être une tâche extrêmement compliquée, voire impossible dans certains cas. Le processus permettant une telle tâche est appelé Vérification & Validation (V&V). Comme résumé par Collofello [20], les approches principales de V&V sont les suivantes :

— Revue de code
— Test logiciel
— Vérification de programme
— Simulation et prototypage
— Suivi des exigences

Chaque approche V&V peut être analysée sous deux aspects. Comme Sannier propose de nous l'expliquer [24], le premier aspect se trouve au niveau du système et le second se concentre sur les spécifications des exigences. Du point de vue du système, les étapes de **vérification** vont s'assurer que le produit en développement satisfera les attentes des parties prenantes et la **validation** sera centrée sur les procédures et la validité du produit final.

2.1.1 Revue de code

La révision technique implique le fait qu'un ou plusieurs spécialistes parcourent le code pour l'auditer. Le but de telles révisions de code est principalement de vérifier que les concepts techniques sont bien respectés. Il ne s'agit pas d'une vérification du code de bas niveau en tant que telle [14].

2.1.2 Test logiciel

Il s'agit de l'action qui vérifie que les exigences soient satisfaites et que les résultats produits soient bien ceux escomptés. Collofello [20] nous propose de distinguer quatre niveaux de tests ; en partant d'un point de vue bas niveau à un point de vue plus haut niveau.

— Le test **unitaire** est le test de niveau le plus bas. Il s'agit en l'espèce de pièces qui composent le logiciel qui sont testées pour s'assurer qu'elles respectent leurs propres spécifications.

— Le test d'**intégration** est réalisé pour avoir la certitude qu'un groupe de composants issus du système soit capable de fonctionner correctement lorsque ces composants travaillent ensemble.

— Le test **système** (ou test **fonctionnel**) est la phase où le système (hardware et software) est testé d'une manière telle que ses exigences espérées seront mises à l'épreuve.

— Les tests de **régression** sont des tests qui seront exécutés durant la phase de développement, principalement pour vérifier qu'aucun nouveau changement n'affecte une partie du programme qui fonctionnait comme espéré avant la mise à niveau du code.

Dans cette étude, une attention tout particulière est portée au test **système**. En effet, lorsqu'on souhaite tester le système comme le ferait un utilisateur qui se sert de l'interface mise à disposition, c'est ce type de test qui est impliqué. De manière générale, ces tests peuvent être construits manuellement ou en suivant une méthodologie particulière. Par exemple, une méthode propose que les tests soient dérivés de scénarios d'utilisation (*Use Cases*). Mais cela implique toujours une lourde charge de travail humain dans la conception des tests. Le TDM est une de ces méthodologies qui permet une automatisation de la génération de cas de tests (CTAs). Dans la section 2.2, nous nous focaliserons entièrement sur ce sujet.

2.1.3 Vérification de programme

Aussi connue sous le nom de "vérification formelle" ou "technique des preuves", il s'agit ici d'une méthode mathématique qui vérifie de manière complète la consistance entre une spécification rigoureuse et mathématique du système et sa solution algorithmique. L'implémentation est donc vérifiée en se basant sur la spécification. Les problèmes de cette méthode sont les suivants :

— La spécification doit être totalement formelle.
— Faire les preuves de programmes très vastes est dans certains cas de figure impossible. Elles peuvent amener à des explosions combinatoires.

2.1.4 Simulation et prototypage

Pour pouvoir analyser le comportement du système durant sa phase de développement, le prototypage est un compromis correct qui permettra à l'utilisateur

de vérifier que les exigences sont respectées. Il peut aussi être utilisé très tôt dans le développement pour analyser les performances espérées. Un point à prendre en considération est que les prototypes utilisés doivent être aussi corrects que possible ; cette méthode nécessite donc son propre processus de V&V.

2.1.5 Suivi des exigences

Utilisé pour s'assurer que le produit ainsi que ses tests satisfont leurs exigences, par exemple en utilisant différents types de matrices de tracabilité [20].

2.1.6 En résumé

Nous nous intéresserons dans la suite de ce travail à la méthode de test fonctionnel. Cette méthode, si elle est utilisée de manière judicieuse, permet de tester tous les composants et artéfacts du système. Chaque entité de test fonctionnel sera considérée comme un scénario d'utilisation possible où l'utilisateur devra interagir avec le système en vue d'effectuer une action complète dessus. Au travers de ces interactions, les modules internes du système seront implicitement testés. Pour être exhaustive, cette méthode doit être utilisée avec assez de cas de tests pour couvrir tous les possibilités envisageables. Sur certains types de sytèmes, la quantité de cas de tests explose et la méthode manuelle ne suffit plus. Une solution pour pallier ce souci est de limiter l'intervention humaine dans l'exécution/la réalisation des tests grâce à l'automatisation de tests.

2.2 Test Dirigé par Modèle (TDM)

Dans une de leurs études, Apfelbaum *et al.* [17] soutiennent l'idée que le TDM devrait être utilisé pour la génération de cas de test. Leur principal argument est que la modélisation est une pratique déjà bien utilisée dans les entreprises logicielles. Ces dernières peuvent déjà s'en servir, par exemple, pour vérifier les attentes des parties prenantes. Il n'y a donc pas de raison de ne pas utiliser la modélisation pour faire d'autres tâches, dont ce qui nous intéresse ici : les tests.

La modélisation permet de concentrer toutes les connaissances sur le fonctionnement du système à un unique endroit (sur le modèle). Elle permet aussi de faire évoluer facilement ce modèle de manière simple si une nouvelle fonctionnalité est ajoutée au système. Du point de vue d'un testeur, ces modèles sont une parfaite source de données puisqu'ils montrent comment le système est supposé réagir aux stimulis et sous quelles conditions exactes.

En comparaison à cela, les définitions textuelles d'un système peuvent être réellement ambigües car le langage humain n'est pas un langage formel et pourrait donc être interprété de plusieurs manières différentes. Prenons l'exemple de la phrase suivante : "L'utilisateur ne dispose que de trois tentatives pour s'identifier dans le système". Le testeur qui doit créer des cas de tests ne peut pas comprendre ce qui se passe si on dépasse les trois tentatives. Avec ce type de spécifications textuelles, les testeurs attendront plutôt que le système soit utilisable pour pouvoir observer comment il réagit aux stimulis car ils n'ont pas reçu de documents le spécifiant de manière formelle. Cette manière de faire ne les assure pas que le comportement du système est le bon ; elle vérifie uniquement qu'il réagit comme le codeur l'a conçu ! On pourrait donc se trouver dans une situation où certaines fonctionnalités sont non couvertes par le jeu de tests réalisé.

En 2014, Binder *et al.* ont réalisé une enquête auprès de 104 utilisateurs de TDM [18]. Certains résultats ont été reportés ci-après en vue de montrer l'attrait du TDM pour réaliser des tests. La Figure 2.1 montre qu'une grande majorité emploie les TDMs pour faire du test fonctionnel, catégorie de test particulièrement visée dans mon travail. Ensuite, comme le montre le diagramme de la Figure 2.2, la majorité des utilisateurs des TDMs sont favorables à l'utilisation de modèles dans d'autres phases du développement. Cela ne nous indique cependant pas si ce sont les mêmes modèles qui sont réutilisés ou si ce sont des tâches redondantes. Cela montre en tout cas un attrait envers la modélisation. Enfin, dans la Figure 2.3, les machines d'état ressortent, de peu certes, favorites de cette enquête.

A la vue de ces résultats, on peut supposer que le développement de modèles via des diagrammes d'état est cohérent. Ce choix de diagramme ne sera pas non plus remis en question dans ce travail puisque notre but premier est de pouvoir proposer un outil de concrétisation qui peut travailler avec un générateur de CTAs indépendant. Nous avons choisi comme générateur VIBeS [16]. VIBeS est un outil créé par l'Université de Namur, qui génère des tests abstraits via une méthologie TDM se basant sur les machines à états.

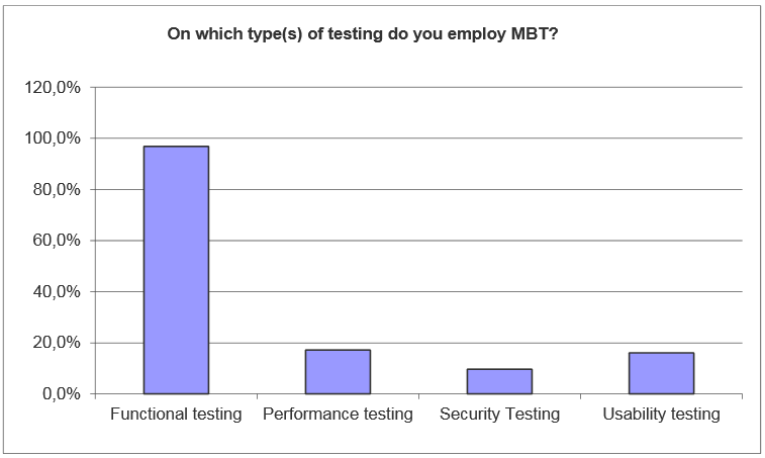


FIGURE 2.1 – Type de test [18]

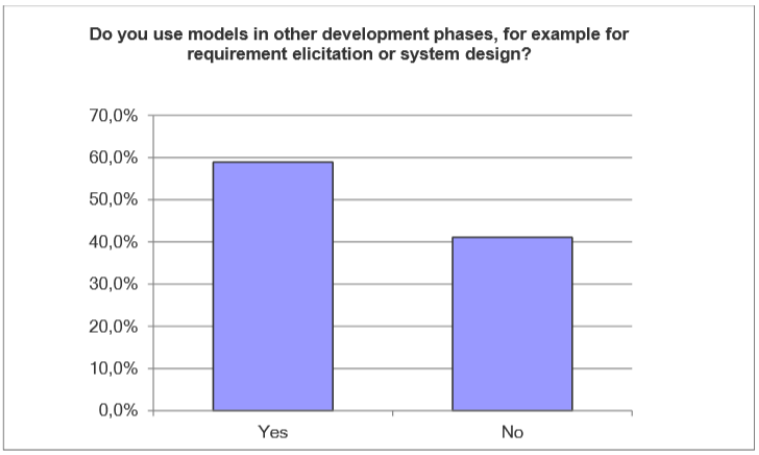


FIGURE 2.2 – Utilisation des modèles [18]

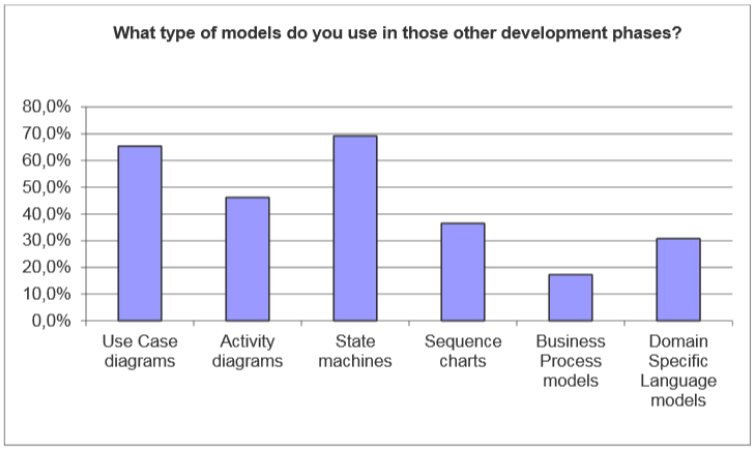


FIGURE 2.3 – Modèles utilisés lors d'autres phases du cycle de développement [18]

2.2.1 Construction du modèle

Selon Apfelbaum [17], les ingénieurs logiciel ont, depuis longtemps, eu recours aux modèles, persistants ou non, pour pouvoir communiquer des idées. Proposer de modéliser l'entièreté du système n'exige donc pas l'acquisition de nouvelles compétences. Toujours dans le même article, l'auteur promeut l'utilisation des machines à états pour représenter de tels comportements ; le même type de diagramme choisi au point précédent. Ce type de modélisation permet d'illustrer la notion d'action faisable qui amène à un résultat ; et où l'action peut être quelque chose de propre à l'état courant ou bien une action générique à toutes les situations.Voici les hypothèses faites sur les diagrammes d'état que nous traiterons directement dans l'outil (aussi sur la Figure 2.4) :

— Chaque état contient un transition entrante et sortante.

— Une transition est empruntée si sa garde est vraie et elle peut exécuter une "output" (ou "action") sur le système.

— Le diagramme doit posséder un état initial et un état final.

— Etant donné un état initial, il n'existe pas d'état mort atteignable depuis cet état. On entend par "état mort", un état à partir duquel il est impossible de sortir.

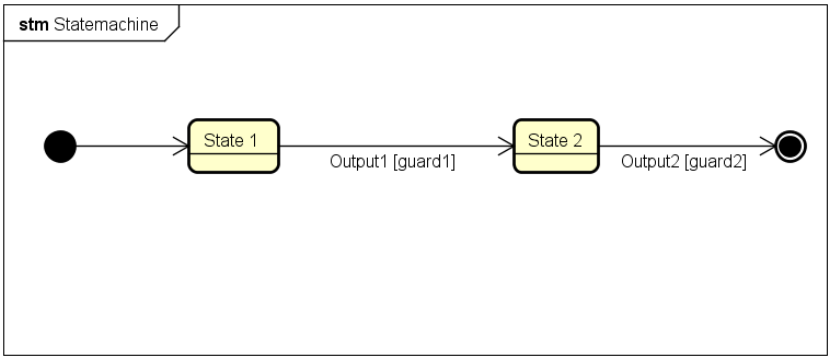


FIGURE 2.4 – Diagramme d'état de base

Réaliser ce genre de modélisation du système, même tardivement sera toujours bénéfique [28]. Elle permettra notamment de mettre en évidence des défauts de conception (qui seront visibles lors de la conception du diagramme), et comme dit précédemment, elle pourra être réutilisée pour créer les scénarii de tests.
Une fois le modèle du système à tester construit, il peut être envisagé d'en extraire les tests. L'objectif est de fournir des scénarii d'utilisation qui montrent que le système fonctionne correctement. Par scénario d'utilisation, on entend "une suite d'actions exécutées sur le système dans le but d'atteindre un objectif déterministe sur ce même système". Ces actions effectuées sont des actions "primitives". Chaque action primitive est la décomposition la plus petite d'une action telle que si on la décomposait encore plus, ses sous-composantes ne seraient jamais utilisées séparément.

2.2.2 Actions primitives

Apfelbaum [17] décrit les actions primitives comme s'intégrant dans au moins une des catégories suivantes :

— **Générer un stimuli sur le système** : ce type d'événement permet de passer d'un état à un autre dans le système. Cette action doit pouvoir être contrôlée (exécutée et validée) depuis l'environnement de test. Un stimuli sur un système tel qu'un site web pourrait, par exemple, être "se logger" ou "afficher le menu principal".

— **Vérifier que le système se comporte correctement** : une vérification de l'état courant du système est faite par rapport à un état prédéterminé. Lorsque le test est effectué sur un système logiciel, l'état courant pourra typiquement être vérifié via du texte ou une image affichée à l'écran. Néanmoins, ce type de vérification n'est pas toujours nécessaire car le simple fait qu'un stimuli ait pu aboutir signifie (dans certains cas) que le système était bien dans l'état espéré.

— **Mise en place de l'environnement** : parfois, le système à tester doit se trouver dans un état prédéterminé pour qu'un test spécifique puisse s'exécuter. Pour ce faire, une action préalable sur le système est nécessaire pour que le test puisse se jouer dans un environnement toujours identique. Si nous prenons l'exemple d'un réseau social, on pourrait imaginer que le scénario consiste à rechercher une publication et de la *liker*. La mise en place de l'environnement consisterait à poster en amont du test ladite publication à retrouver. D'autre part, on pourrait tout aussi bien imaginer qu'un système ait besoin qu'un logiciel tiers soit démarré pour pouvoir fonctionner, et dans ce cas, la mise en place de l'environnement demanderait de lancer ce dernier.

— **Enregistrer les résultats** : en partant du principe que les tests seront entièrement automatisés, il faut avoir une trace de toutes les exécutions sur le système visé. Chaque action, réussie ou échouée, devrait donc pouvoir être analysée par la suite. Une interface doit idéalemenent exister pour rendre cette analyse la plus facile possible.

Dans notre approche, les choses seront légèrement différentes par rapport à ce qui est proposé ci-dessus. Deux des quatres catégories d'actions primitives seront, partiellement, sorties du modèle pour être prises en charge par le *framework* de test :

— **Les stimulis** seront les actions définies pour naviguer d'état en état, sur le modèle de base, ou dans le test dérivé. Elles seront appelées **actions** ou **stimulis**

— **Les assertions** seront les vérifications de comportement qui pourront être faites lorsqu'on rentre dans un état. Elles seront appelées **asserts** ou **assertions**

— **La mise en place de l'environnement** qui est une action réalisée sur le système pour le mettre dans un état déterminé pour effectuer un test sera considérée comme un stimuli normal sur le système. Une action qui est une

mise en place d'un état pré-requis de **l'environnement de test** ne sera pas intégrée dans le modèle mais sera sous la responsabilité de cet environnement.

— **Enregistrer les événements** qui se produisent lors de l'exécution d'un test sera sortie du modèle et sera directement intégrée au *framework* de test.

Un test extrait du modèle sera donc un scénario complet, qui commencera de l'état initial, qui exécutera des stimulis qui l'entraîneront dans des états dans lesquels il pourra être soumis à des assertions ; et tout cela en empruntant un seul chemin. Un exemple concret sera développé dans la sous-section **??**

La Figure 2.5 montre notre précédent diagramme d'état mis à jour pour prendre en compte les dernières considérations.

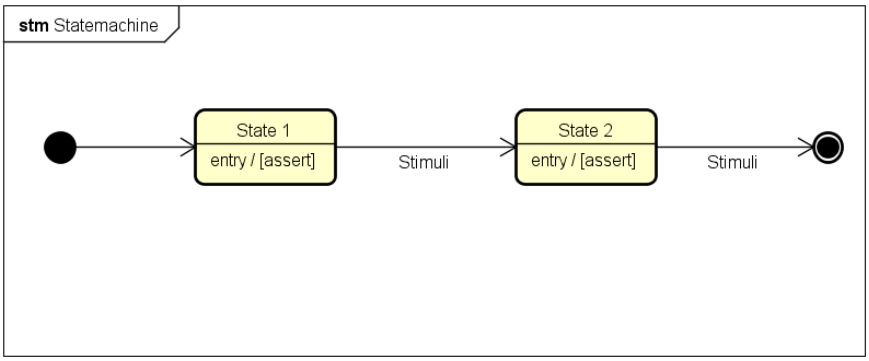


FIGURE 2.5 – Diagramme d'état revu

2.2.3 Extraction des tests et critères de couverture

Lorsqu'on parle de couverture, on parle de la manière dont on recouvre les différentes parties d'un système modélisé. Plusieurs types de critères existent pour réaliser cette couverture ; ce sont les critères de couverture. Certains d'entre eux [26],sont orientés uniquement sur le graphe et non sur le système, ou encore sur les données utilisées par le système. Les points suivants montreront comment nous pouvons recouvrir un modèle en se basant sur le graphe. Ces critères de couverture [23] peuvent nous donner une indication sur l'acceptabilité de nos tests issus du modèle. En effet, si la couverture n'est pas suffisante, les tests pourront ne pas révéler les problèmes potentiels. Cette tâche d'extraction des tests à partir du modèle du système sera réalisée par l'outil utilisé en amont de notre concrétisateur ; donc ici, par VIBeS. Mais puisque nous en manipulerons ici, voici les principaux critères de couverture de graphes utilisés dans ce travail.

2.2.3.1 Couverture de tous les noeuds

Ce critère a pour but de couvrir tous les états du modèle. On évalue la qualité de la couverture avec la formule : **nombre de noeuds couverts / nombre de**

noeuds total.

Taktak [26] évoque le danger de ce type de couverture en prenant pour exemple un modèle représentant un bout de code. Dans certains cas, on pourrait passer par tous les noeuds sans pour autant emprunter des transitions qui révèleraient des erreurs. Ce cas précis n'est pas trop inquiétant dans notre cas d'utilisation. Il ne se reproduira pas tel quel car nos tests seront de beaucoup plus haut niveau. Par contre ignorer des transitions qui sont donc des stimulis, est en effet une possibilité d'ignorer des erreurs probables. Si on suit ce critère sur la Figure 2.6, le seul cas de tests nécessaire est :(début,1,a,2,b,3,c,4,e,fin)

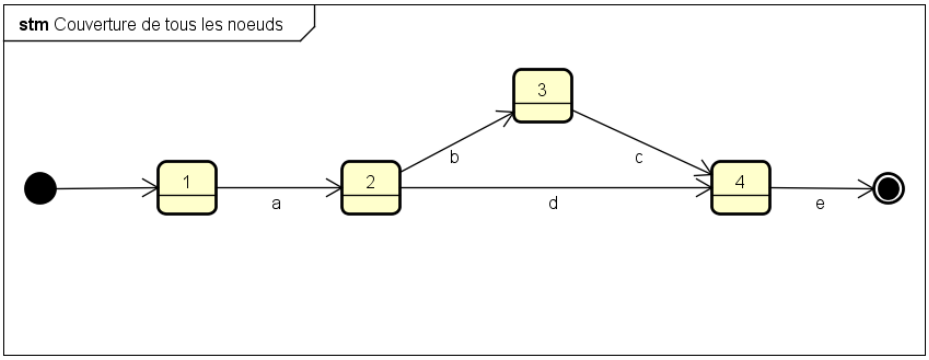


FIGURE 2.6 – Couverture de tous les états

2.2.3.2 Couverture de tous les arcs/transitions

Le but de ce critère est de couvrir un maximum de transitions sur le modèle. Son efficacité peut être évaluée par la formule : **nombre de transitions couvertes / nombre total de transitions**.

Dans notre cas, lorsqu'on évoque les transitions, on parle des stimulis. Ce critère permettra donc de tester tous les stimulis possibles. Et comme chaque stimuli amène à un état, tous les états seront aussi empruntés. Selon ce critère, nous pouvons extraire de la Figure 2.7, les différents cas de tests :

— (début,1,a,2,b,3,c,5,g,fin)
— (début,1,a,2,e,4,f,5,g,fin)
— (début,1,a,2,d,5,g,fin)

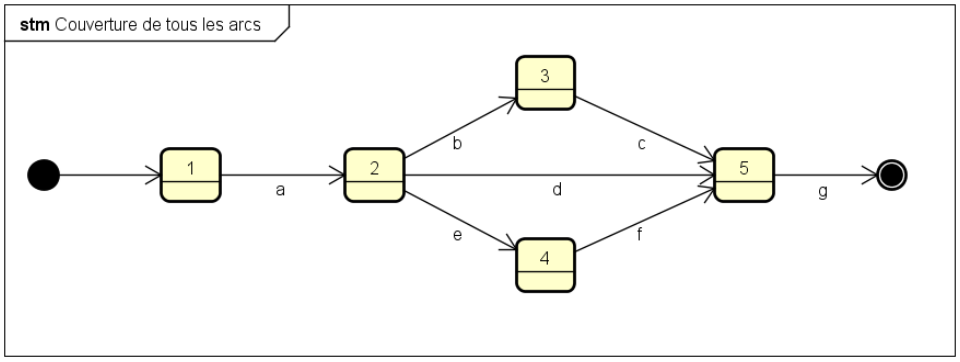


FIGURE 2.7 – Couverture de toutes les transitions

2.2.3.3 Couverture de tous les chemins

Tous les chemins **faisables** seront empruntés. Les mêmes stimulis seront donc empruntés une multitude de fois. Si il est respecté, ce critère passera par tous les états et toutes les transitions. Par contre, il amènera bien souvent un bien plus grand nombre de testscomme le montre la Figure 2.8 :

— (début,1,a,2,b,3,c,5,g,6,i,fin)
— (début,1,a,2,b,3,c,5,h,7,j,fin)
— (début,1,a,2,d,5,g,6,i,fin)
— (début,1,a,2,d,5,h,7,j,fin)
— (début,1,a,2,e,4,f,5,h,7,j,fin)
— (début,1,a,2,e,4,f,5,g,6,i,fin)

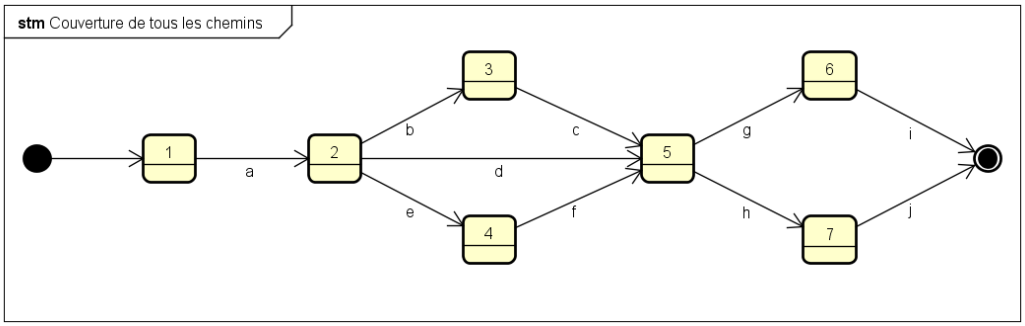


FIGURE 2.8 – Couverture de tous les chemins

2.2.4 Cas d'étude : le réseau social simplifié

Pour illuster la modélisation et l'extraction de tests, nous allons imaginer un système à tester relativement simple : un réseau social simplifié. Nous considèrerons qu'il permet de se connecter, de créer des statuts, et enfin de "liker" ces statuts. En premier lieu, nous avons donc défini sur la Figure 2.9 le modèle du réseau social simplifié :

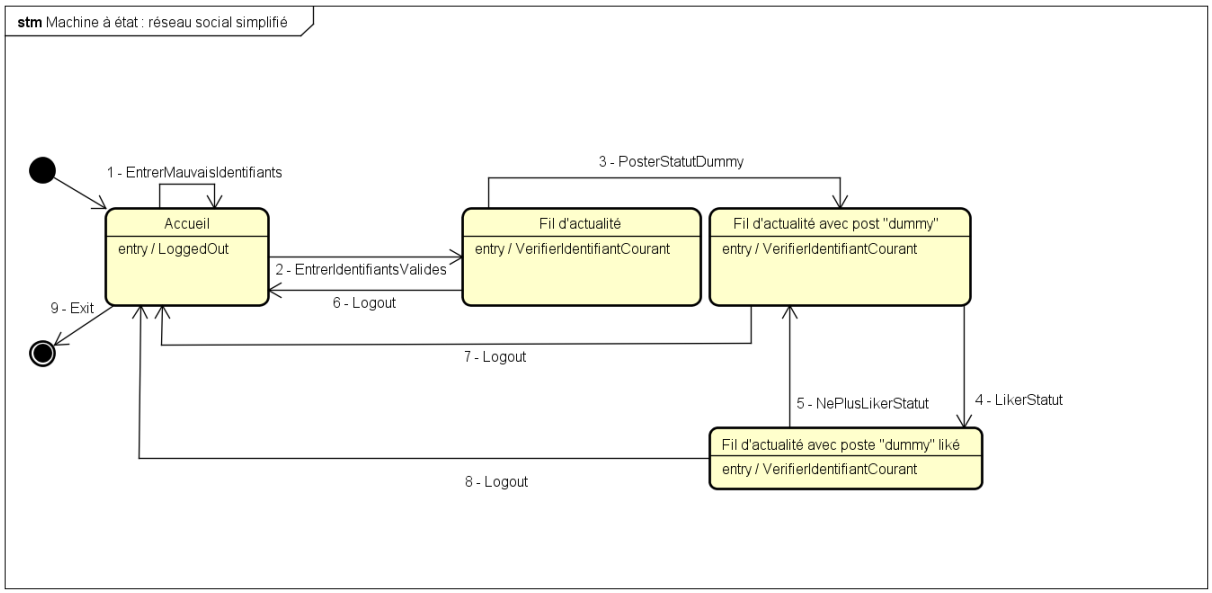


FIGURE 2.9 – Diagramme d'état de notre réseau social simplifié

Ensuite, il s'agit d'en extraire des tests comme expliqué précédemment. Ici on choisira d'emprunter au moins une fois chaque stimuli, comme illustré sur les figures suivantes :

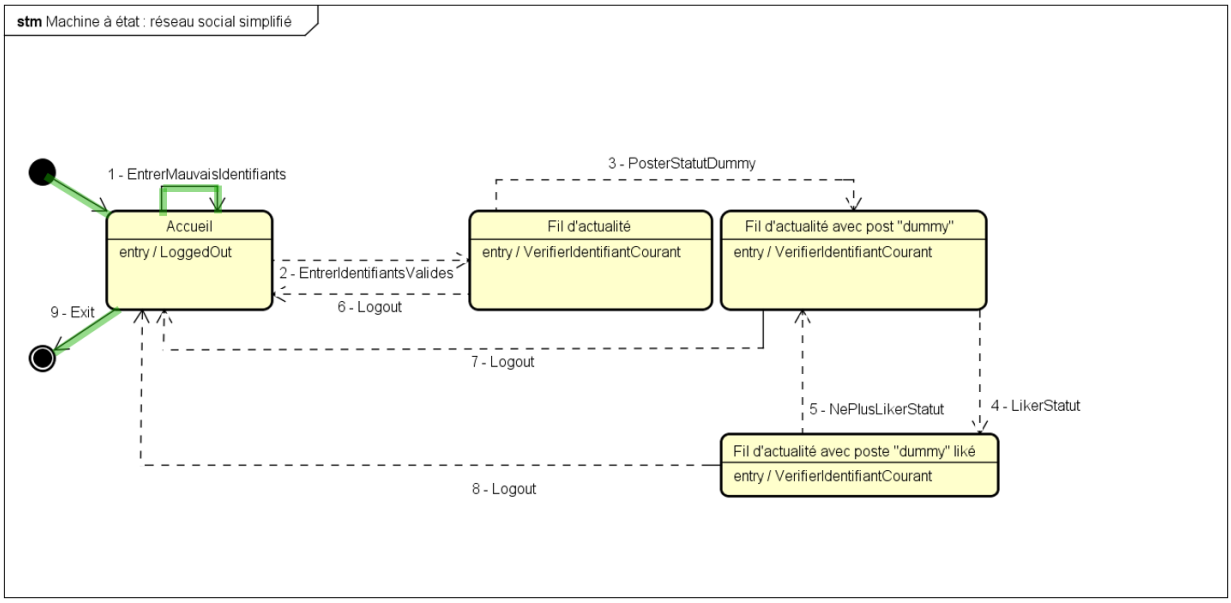


FIGURE 2.10 – Cas de test 1 : séquence 1-9. Tenter des mauvais identifiants

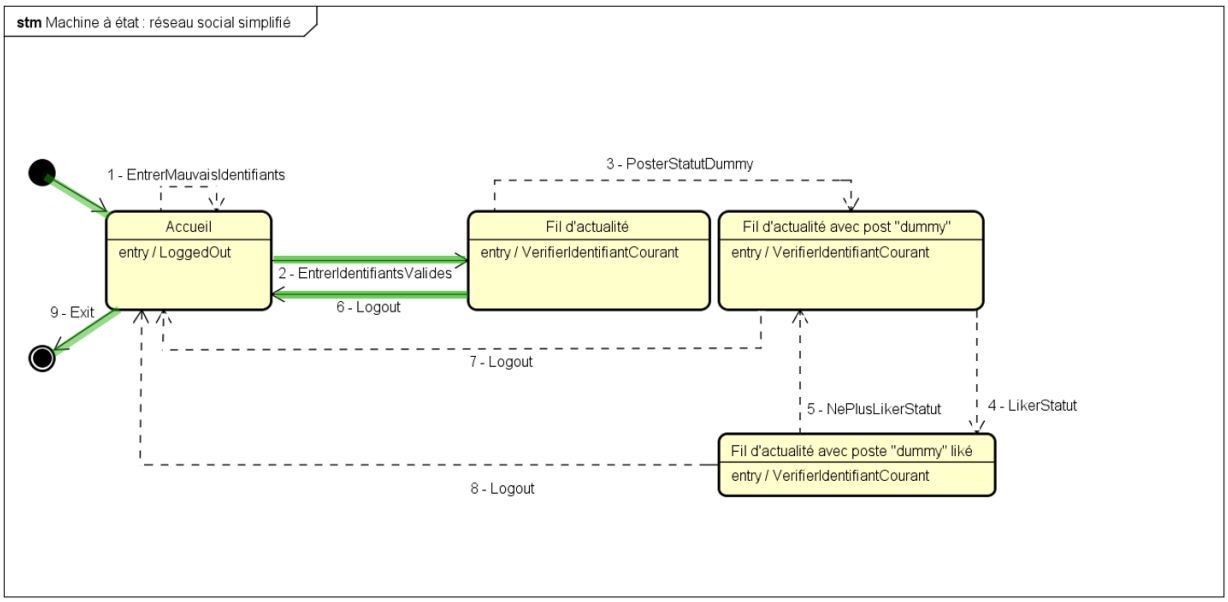


FIGURE 2.11 – Cas de test 2 : séquence 2-6-9. S'enregistrer puis quitter la session

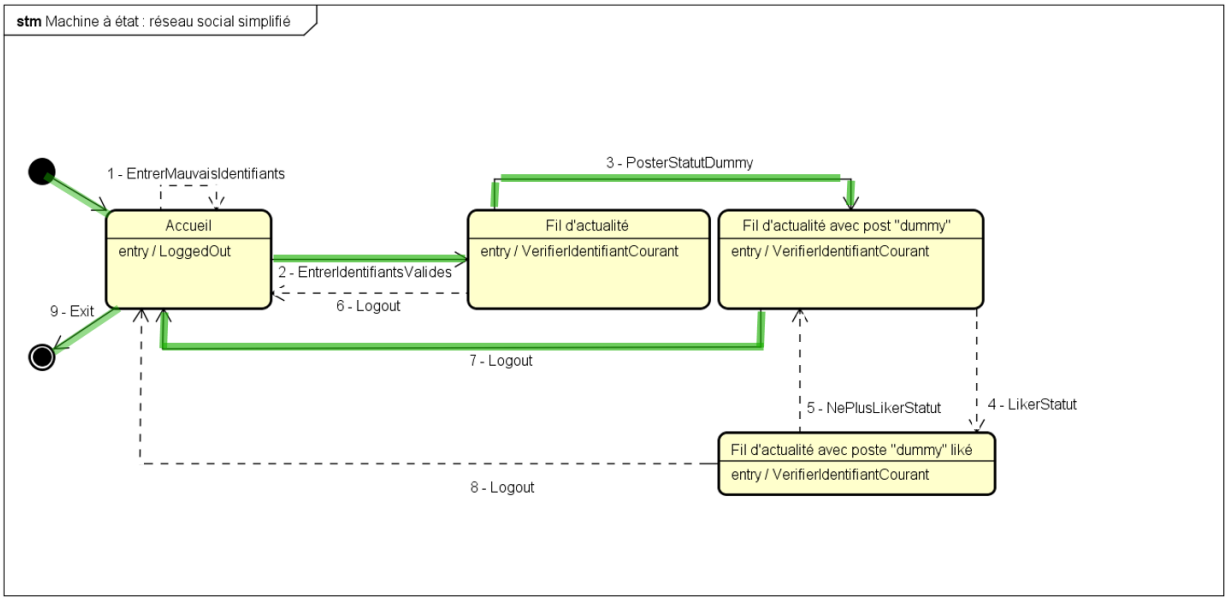


FIGURE 2.12 – Cas de test 3 : séquence 2-3-7-9. Une fois enregistré, poster un message. Ensuite quitter la session

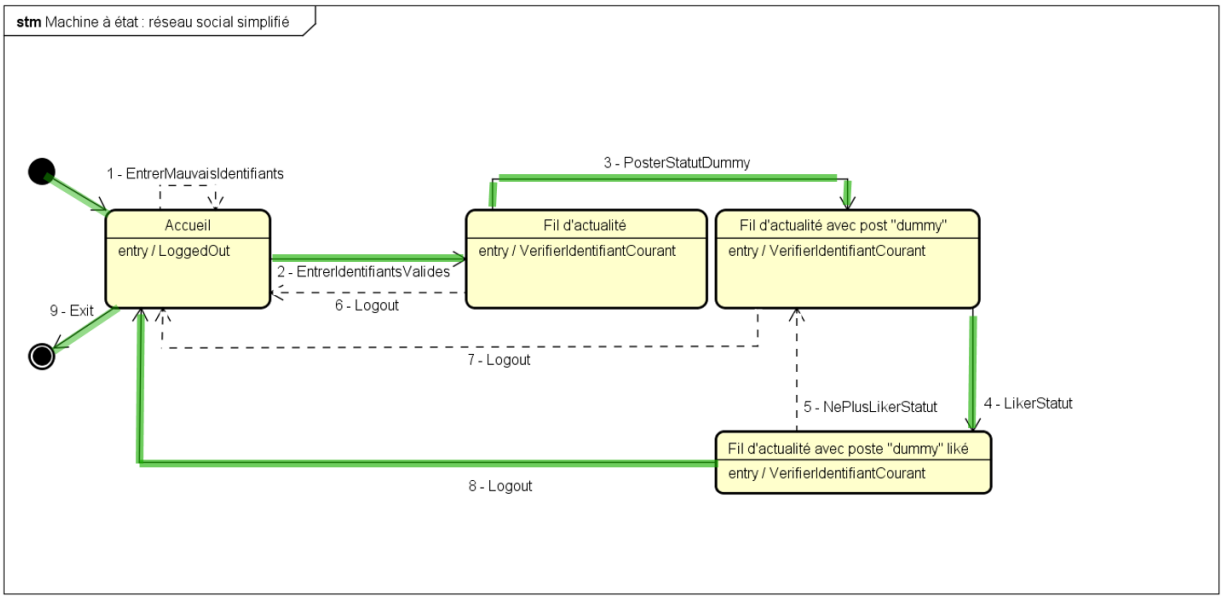


FIGURE 2.13 – Cas de test 4 : séquence 2-3-4-8-9. Une fois enregistré, poster un message et le "liker". Ensuite quitter la session

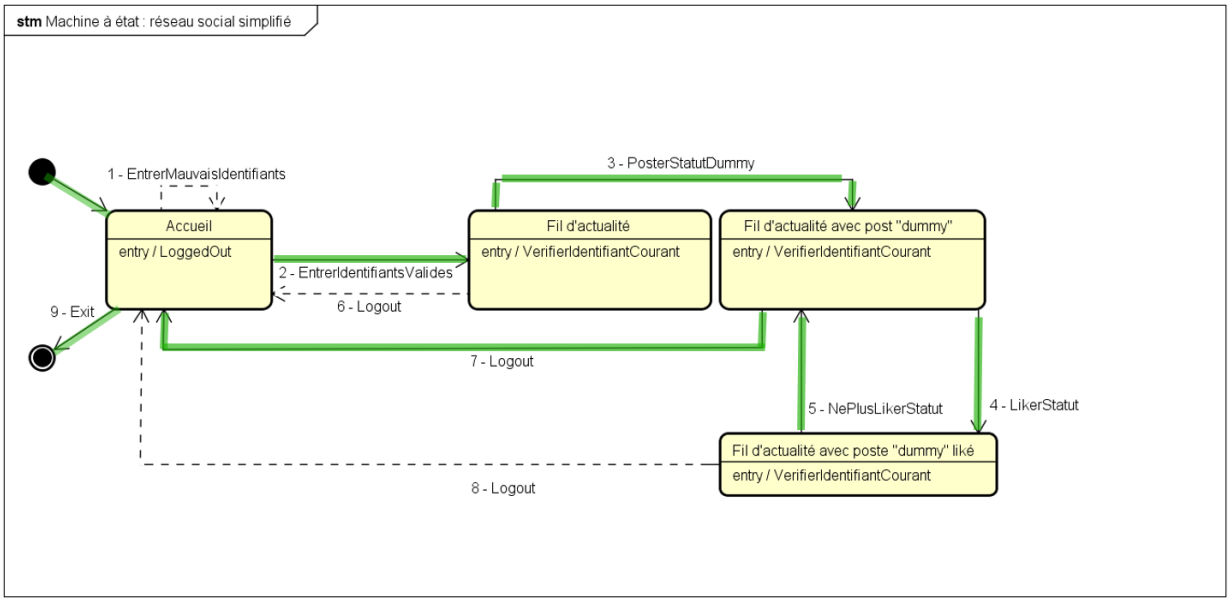


FIGURE 2.14 – Cas de test 5 : séquence 2-3-4-5-7-9. Une fois enregistré, poster un messager, le "liker" et après, retirer ce *like*. Ensuite quitter la session

On peut observer que sur un système extrêmement simplifié, le nombre de tests croît rapidement. De plus, ici, la partie concernant les données a été abstraite ; c'est-à-dire que nous ne savons toujours pas quels seront les identifiants testés, les statuts postés, etc... Le fait de vouloir convertir ces tests de haut niveau en scripts et de pouvoir les faire s'exécuter seuls est réellement justifié. Exprimés comme tels, ils peuvent être joués manuellement par une personne sans qualification nécessaire, par contre, pour pouvoir les convertir en code exécutable, on voit très vite qu'il nous manque une série d'informations :

— Les valeurs à utiliser
— Le code exécutable correspondant aux stimulis
— Le code exécutable correspondant aux assertions
— L'accès à l'interface graphique/physique du système à tester

Dans le point suivant, nous présenterons des outils existants qui pourraient aider à la concrétisation/exécution automatique de tests.

2.3 Concrétisation de tests automatisés

Vouloir automatiser des tests, n'est, à l'heure actuelle, pas une idée révolutionnaire. Que du contraire : des dizaines d'outils existent pour automatiser l'exécution des tests. Le principal souci est que ceux-ci sont souvent dédiés à une technologie donnée et ne sont donc plus utilisables lorsqu'on change d'environnement : il existe d'excellentes solutions pour tester des logiciels via leur interface développée en .NET, en C++, en HTML ou d'autres, mais peu, voire pas, de propositions permettent de tester tous ces types d'interface et ce, d'une manière uniforme du point de vue de l'ingénieur de tests. Différents outils sont présentés ci-après ; ils ont été choisis car ils possèdent au moins un atout voulu ou une caractéristique que nous voulons absolument éviter. Ensuite, nous ferons un tour d'horizon sur les solutions imaginées par des chercheurs répondant à la même problématique que celle traitée dans cette étude, à savoir, la concrétisation de test abstraits.

2.3.1 Outils pour l'automatisation de tests

Certains outils permettent d'automatiser des actions sur un système. Certains d'entre-eux demandent à l'utilisateur d'écrire des scripts dans un langage donné et d'autres demandent par exemple de réaliser manuellement les actions pendant que le logiciel les enregistre. Les outils ne font rien de plus qu'exécuter ce que l'utilisateur aura défini.

2.3.1.1 Selenium Web Driver

Selenium [9] est un outil extrêmement populaire à l'heure actuelle. Ce *web driver open source*, permet d'exécuter des actions prédéfinies sur un navigateur web. Son implémentation ouverte a rendu possible l'écriture de ses tests dans une multitude de langages. Ce qui en fait donc un solution idéale pour ce genre de tests. De plus, certains outils auxiliaires ont été développés pour pouvoir enregistrer des séquences de tests réalisées par un opérateur et de pouvoir les rejouer par la suite. C'est actuellement une référence dans le domaine des tests d'interface web. Nous retiendrons pour Selenium :

+ Il offre énormément de méthodes pour tester des applications web.
+ Il est *open source*.
+ Il permet l'écriture des tests dans un langage de programmation parmi une mutltitude de propositions.
− Il se focalise uniquement sur les applications web.

2.3.1.2 Sahi

Tout comme Selenium, Sahi [8] permet de tester des applications Web. Un peu moins populaire, il propose néanmoins les mêmes fonctionnalités et même plus. Son implémentation plus fermée oblige l'utilisateur à exécuter les scripts écrits dans un environnement dédié. Cela rend malheureusement l'intégration de Sahi dans un projet de plus grande envergure bien plus complexe. Nous retiendrons pour Sahi :

+ Il offre énormément de méthodes pour tester des applications web.
− Il est n'est pas *open source*.
− Il impose la manière d'écrire les tests.

2.3.1.3 AutoHotKey

AutoHotKey [2] (aussi appelé "AHK") est tellement vaste qu'on pourrait parler d'un langage de programmation destiné à la réalisation de tests et pas simplement d'un outil d'exécution de tests. Initialement créé pour que l'utilisateur puisse répéter des simulations de frappes au clavier, ses possibilités ont été largement étoffées. En fait, AHK permet d'automatiser des tâches sur windows ou sur un programme (tournant sur windows). AHK propose aussi la reconnaissance d'images, ce qui permettra de lui indiquer à l'aide de captures d'écran les zones auxquelles il doit accéder. Si AHK a un champ d'action très large, c'est en partie un défaut car il ne propose pas de méthode efficace pour tester des interfaces web, .NET, C ou autres. C'est dans beaucoup de cas, quelque chose de faisable, mais le résultat dépendra uniquement de l'implémentation réalisée via le langage proposé. De plus, pour chaque logiciel il faudra développer un nouveau code qui permettra de le tester. Enfin, comme cité plus haut, AHK ne peut tourner que sur Windows. Nous retiendrons les différents avantages et inconvénients suivants :

+ Il permet de faire de la reconnaissance d'images pour trouver des zones à l'écran.
+ Il permet d'aller très loin dans la création de tests du fait qu'il s'agit d'un langage.
− Il ne peut compiler ses scripts que sur windows.
− Il offre une efficacité fortement dépendante de la personne qui code.

2.3.1.4 Sikuli

Si comme AHK, Sikuli [10] supporte la reconnaissance d'images, dans le cas présent, c'est son mode de fonctionnement principal. Cet outil a l'avantage de pouvoir ignorer le langage dans lequel l'interface utilisateur a été développée. Cela comporte tout de même deux désavantages : tout d'abord, l'interface est le composant le plus suceptible de changer dans la vie du logiciel, et donc, cela signifierait de devoir recommencer les scripts de tests automatisés. Le deuxième, et pas des moindres, est que nous sommes très liés à un environnement de test en particulier. En effet, selon la résolution de l'écran, la version ou la marque du navigateur web, les images ne seront pas reconnues et les tests seront inutilisables. Nous en retiendrons :

+ Il permet de tester une multitude de types de *GUI*.
− Il impose la manière d'écrire les tests.

2.3.1.5 Squish GUI tester

Logiciel payant, Squish [11] se dit compatible avec un majorité d'interfaces utilisateur. De nouveau, l'idée du créateur est de fournir une solution plus fermée et bien encadrée qui rend toute intégration dans un autre projet plutôt complexe. De plus, Squish base en grande partie le développement des scripts de tests sur le *play & replay*, ce qui signifie que le logiciel ne propose pas un langage accessible pour l'écriture des tests automatisés comme nous en avons besoin dans le cas présent. Nous retiendrons de cet outil :

+ Il permet de tester une multitude de type de *GUI*.
− Il impose la manière d'écrire les tests.

2.3.2 *Frameworks* existant pour la concrétisation de tests

Les *frameworks* présentés ici sont différents des logiciels proposés ci-dessus. Ces derniers permettent d'écrire des tests qui s'exécutent seuls par la suite. Les *frameworks* de concrétisation proposent une méthode et des outils permettant de partir de tests abstraits pour obtenir des tests automatisés.

2.3.2.1 STALE

Li *et al.* [22] décomposent leur explication en deux parties. La première explique comment le *framework* proposé peut créer des tests concrets sur base d'un modèle du système. Si nous aussi nous souhaitons utiliser le TDM, dans notre solution nous souhaitons justement être un maximum indépendant du générateur de CTAs, quelle que soit la méthode utilisée, tant que la sortie est formatée comme on l'attend.
La deuxième partie explique une méthodologie et un algorithme pour convertir des tests abstraits en tests concrets exécutables de manière automatique. Le tout forme un *framework* nommé "STALE" pour *Structured Test Automation Language Framework*. Le processus suivi est illustré sur la Figure 2.15.

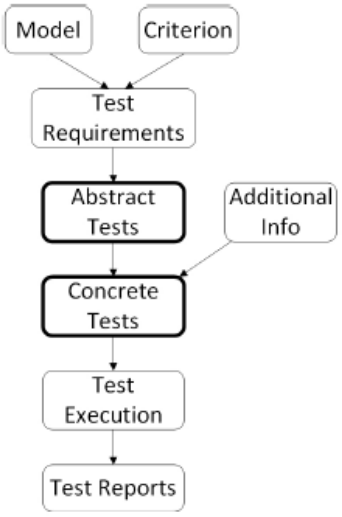


FIGURE 2.15 – Processus générique du TDM [22]

Pour expliquer le fonctionnement de STALE, nous utiliserons l'exemple utilisé par Li *et al.*[22] : le distributeur de friandises. Voici une machine à état complète du système représentée dans la Figure 2.16.

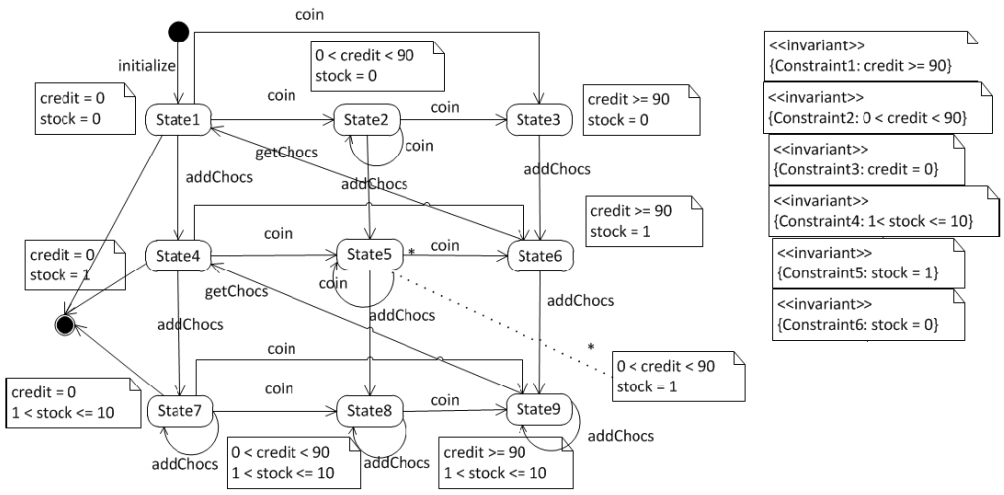


FIGURE 2.16 – La machine à boissons [22]

S'il est évident que le processus attend des tests abstraits pour réaliser leur concrétisation, il est très astucieux d'attendre aussi un *mapping* pour lier la partie abstraite à un code qui pourra exécuter les actions concrètes. Pour ce faire, les auteurs ont imaginé deux types de *mapping* : *object mapping* et *element mapping*. Tandis que les *element mappings* seront plutôt utilisés pour définir le code lié à une transition, l'*object mapping* sera plutôt dédié à la définition et la création des instances qui seront utilisées dans les méthodes définies précédemment. Pour un *mapping* d'élément, un exemple est proposé [22] où on lie la méthode concrète "getChocolate" à la transition "getChocs" :

```
Mapping getChocolate Transition getChocs
{
StringBuffer sb = new StringBuffer(‘‘MM’’);
vm.getChoc(sb); \\vm est l’instance représentant la machine
}
```

La sémantique choisie peut se montrer parfois un peu complexe mais c'est obligatoire compte tenu de la dépendance forte qu'il peut exister entre les différents *mappings* définis. Le contenu que l'on mettra dans ces *mappings* est du code qui sera exécuté par l'environnement de test choisi (non prévu par STALE). En réalité, STALE ne fera qu'une concaténation intelligente des *mappings*. Les auteurs [22] avancent que cette méthode présente un gros avantage car elle réduit le nombre d'erreurs possibles. Si l'outil utilise plusieurs fois la même portion de code écrit une unique fois, cela signifie que l'utilisateur aura moins d'opportunités de faire des fautes de langage.

Pour l'étape de concrétisation, les auteurs proposent un autre aspect intéressant : un *constraint solver* qui permet au testeur d'entrer un mot clé (par exemple : *anyInt*) qui indique à STALE de générer une valeur aléatoire. En effet, lors de l'exécution d'un test, pour certaines variables il faut pouvoir contrôler leur valeur exacte (telle que celle d'un identifiant ou d'un mot de passe par exemple), mais pour certaines autres, on veut justement pouvoir tester les limites d'acceptance du système en générant des valeurs hors du commun pour révéler les possibles failles.

De cet outil nous pouvons retenir du positif et du négatif :

+ Son système de *mapping* permet de diminuer la quantité de code à écrire.

+ Son *constraint solver* permet de générer des valeurs aléatoires mais contrôlées grâce aux différents mots-clés.

– Il ne propose pas de méthode pour décrire l'interface à tester. On y accède directement dans le code fourni.

– Son système de *mapping* est trop complexe : STALE a besoin de connaître les liens existants entre les différents *mappings* fournis car l'algorithme de concaténation en a besoin pour générer un code correct.

– Il fournit donc une aide à la génération de code. Il concrétise du code, mais pas des tests. Il assemble des bouts de code fournis mais ne propose aucun moyen d'exécuter les codes générés !

2.3.2.2 Matelo

Matelo [27] est un outil qui permet de concrétiser les tests issus de la méthodologie TDM. Il permet à l'utilisateur de définir un modèle et de laisser le logiciel en extraire des tests abstraits. Mais il nous autorise aussi à définir des tests abstraits directement comme illustré dans la Figure 2.17.

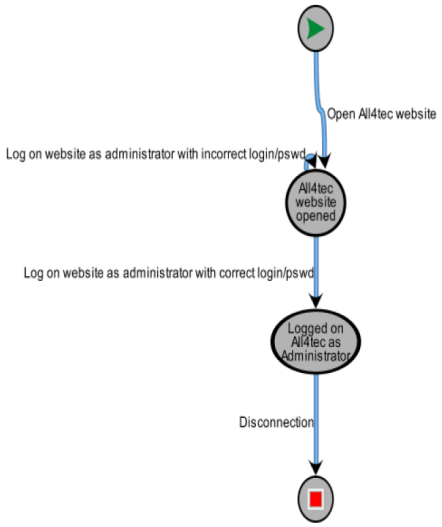


FIGURE 2.17 – Définition d'un test abstrait avec Matelo [27]

Matelo mise sur la concrétisation de tests pour des applications web grâce à Selenium. L'outil ne nous propose donc pas de pouvoir générer des tests concrets pour d'autres types de systèmes.
Pour pouvoir générer du code, Matelo ne demande pas de fichier de *mapping* comme le fait STALE. Il propose , via une interface graphique prévue à cet effet, de faciliter la définition des "stimulis" et des "asserts" comme le montre la Figure 2.18.

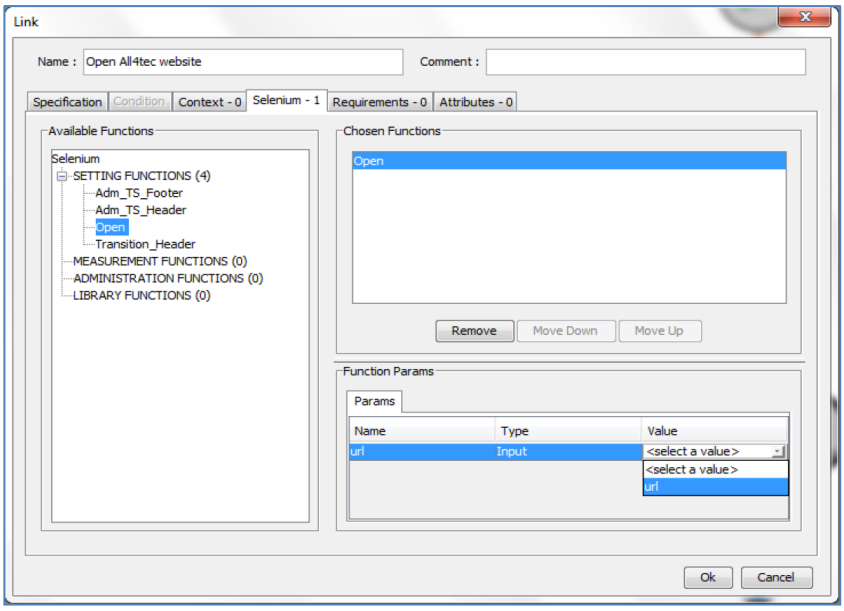


FIGURE 2.18 – Définition d'une transition du modèle [27]

Si cette méthode semble rendre la tâche plus facile, elle est toutefois très dépendante de Selenium. On ne peut uniquement y définir des appels vers des méthodes de ce dernier. Si l'utilisateur souhaite intégrer du code ou un algorithme en plus, ce n'est pas possible. Nous pouvons retenir de ce logiciel :

+ Matelo propose une interface graphique riche et facilitant le processus de définition du contenu des "stimulis" et des "asserts".

+ L'outil permet de démarrer le processus de concrétisation depuis des tests abstraits modélisés ou depuis un modèle complet.

– Matelo n'est ni open-source, ni gratuit. Ces deux points étant des freins pour la combinaison du logiciel avec VIBeS.

– Matelo propose une solution pour les applications web uniquement.

– Il n'est pas possible pour l'utilisateur d'intégrer du code (et donc une certaine forme d'intelligence) dans le contenu des "opérations". Il peut uniquement faire appel aux méthodes proposées par Selenium.

2.4 Résumé

Dans ce chapitre, nous avons tout d'abord énoncé le problème que pouvait représenter les tests dans le monde logiciel d'aujourd'hui. Nous avons ensuite brièvement discuté des TDMs et nous avons expliqué comment cette méthode permet de générer des cas de tests abstraits (CTAs).
Après, un focus a été pointé sur quelques outils existants qui permettent l'automatisation de tests. Si ces outils ne seront pas utilisés **directement**, ils présentent tous des atouts et des défauts non négligeables.

Enfin, une brève description de deux outils permettant la concrétisation de tests abstraits a été réalisée de manière à récupérer quelques éléments sur le sujet et nous permettre de faire des choix judicieux pour la suite de ce travail.

Chapitre 3

Approches et motivations

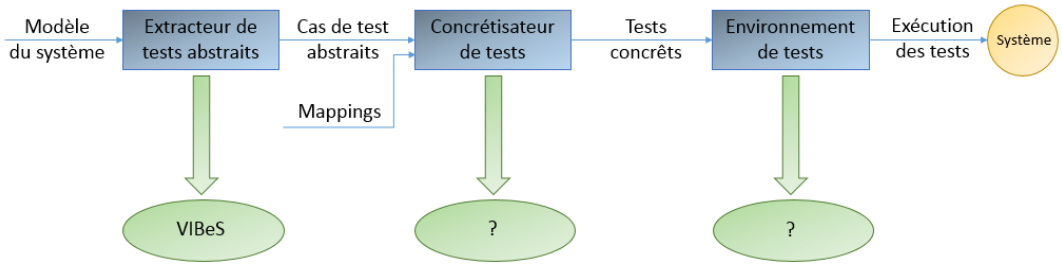


FIGURE 3.1 – Notre chaîne de tests

Dans le chapitre précédent, le problème a été posé et les points sensibles ont été mis en évidence. Dans ce chapitre-ci, nous proposerons des solutions pour pallier ces derniers. Tout d'abord, nous présenterons l'environnement de test. C'est cet environnement de test qui sera utilisé pour développer une proposition de concrétisateur de tests abstraits.
Ensuite, et parce que le but premier de cette étude est de s'interfacer avec VIBeS [16], on passera en revue les enrichissements nécessaires qui ont dû lui être apportés.
Enfin, une solution personnelle sera apportée pour répondre à la problématique de la description de l'interface à tester - étape nécessaire pour pouvoir créer des tests concrets de manière automatisée.
La Figure 3.1 représente l'évolution du problème.

3.1 Environnement de test QTaste

3.1.1 Principe

QTaste [7] est un environnement de test *open source* développé par la société QSpin. Dans un premier temps, cet environnement a été créé pour répondre aux besoins d'un client de la société. Par la suite, vu les possibilités que l'outil offrait d'un côté, et le manque de ressources de l'autre, les sources du logiciel ont été rendues libres. De cette manière, il peut continuer à évoluer sans que l'intervention de la société ne soit nécessaire. C'est ausi grâce à son architecture que QTaste a été retenu pour être la base du concrétisateur de tests présenté. En effet, il propose d'utiliser des *Test APIs* pour communiquer avec le système que l'on souhaite tester, de telle manière que si notre système ne propose pas d'interface compatible avec ce qui existe déjà, il suffit de créer sa propre *Test API*.
Ainsi, le *framework* promet la possibilité d'exécuter des tests sur un environnement *software* ou même *hardware*.

3.1.2 Architecture

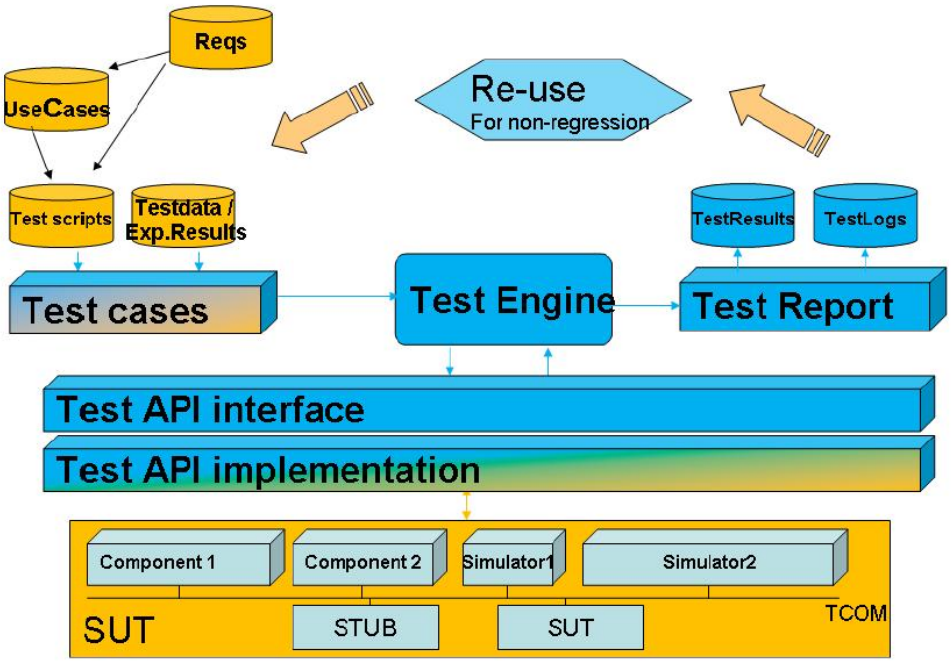


FIGURE 3.2 – L'architecture de QTaste [7]

La Figure 3.2 montre l'architecture de QTaste. QTaste travaille avec des scripts écrits en Python. On peut utiliser les méthodes proposées par le langage de base, ou par des librairies qu'on ajoute. De plus, on peut accéder aux méthodes publiques des *Test API* qu'on utilise. En parallèle à ces scripts, nous avons des données de test ; en effet, le logiciel propose de séparer complètement les deux concepts. Si le test demande d'écrire un mot *MAVARIABLE* dans un champ, et que du côté des données nous avons cinq valeurs différentes pour la variables *MAVARIABLE*, le test s'exécutera cinq fois avec, à chaque exécution, une valeur différente.
Ces scripts de test sont joués par l'engin de test qui lui-même repose en grande

partie sur les *Test APIs* utilisées qui, comme expliqué plus haut, vont communiquer avec le *software/hardware* à tester. Du côté du système à tester, c'est à l'utilisateur d'implémenter ou non des interfaces qui pourront aider à le coupler avec QTaste. Pour les système standards, il est fort probable qu'une *Test API* soit déjà fournie avec QTaste ou soit facilement adaptable.
Une fois l'exécution terminée, un rapport HTML est généré et visualisable directement sur l'application. Ce rapport indique quels tests ont réussi ou échoué, et si tel est le cas, il indique quelle est l'étape du script qui a causé le problème et quelles ont été les valeurs utilisées. Dans le chapitre précédent, nous avions imposé le fait que l'environnement de test devait enregistrer les événements. Ce point est donc validé avec QTaste.
Avant de démarrer une suite de tests, QTaste offre la possibilité de démarrer des logiciels tiers qui sont nécessaires pour la bonne exécution des tests. Comme pour le point précédent, la "mise en place" avait été aussi mise sous la responsabilité du logiciel d'exécution des tests, et c'est donc à nouveau validé pour QTaste.

3.1.3 Test APIs

Les *Test APIs* sont incontestablement la grande force de QTaste. Elles permettent de tester n'importe quel type de système, quitte à devoir écrire nous-mêmes notre propre *Test API* pour le contrôler.
Au niveau des scripts de tests, leur utilisation est relativement transparente. On crée un objet qui instancie la *Test API* souhaitée. Ensuite, dans le reste du script, on peut appeler les méthodes de la *Test API* sur son instance. Les méthodes que l'utilisateur peut appeler se trouvent dans l'aide qui a été créée automatiquement lors du *build* de la *Test API*. En effet, lorsqu'on crée une *Test API* (en java uniquement), on doit créer au moins deux fichiers :

— **Une interface Java** qui définit quelles sont les méthodes publiques auxquelles les scripts peuvent faire appel. C'est aussi ici qu'on définit si la *Test API* pourra être instanciée plusieurs fois ou s'il s'agit d'un "singleton".

— **Une classe Java** qui implémente l'interface définie ci-dessus.

Pour montrer à quel point les *Test APIs* peuvent être un outil puissant, nous allons en illustrer l'utilisation au travers d'un exemple. Prenons un fauteuil dont on veut s'assurer qu'il puisse résister à certaines forces. On utilise un vérin pour simuler une personne qui s'asseoit dessus et ce dernier est connecté à un ordinateur au travers d'un contrôleur USB pilotable en langage Java. Ce genre de test peut être réalisé de manière simple comme montré sur la Figure 3.3. Le test QTaste écrit en Python fait appel aux méthodes définies dans la *Test API ControleVerin* qui elle-même utilise des librairies Java dédiées permettant de communiquer avec la carte de contrôle des vérins.

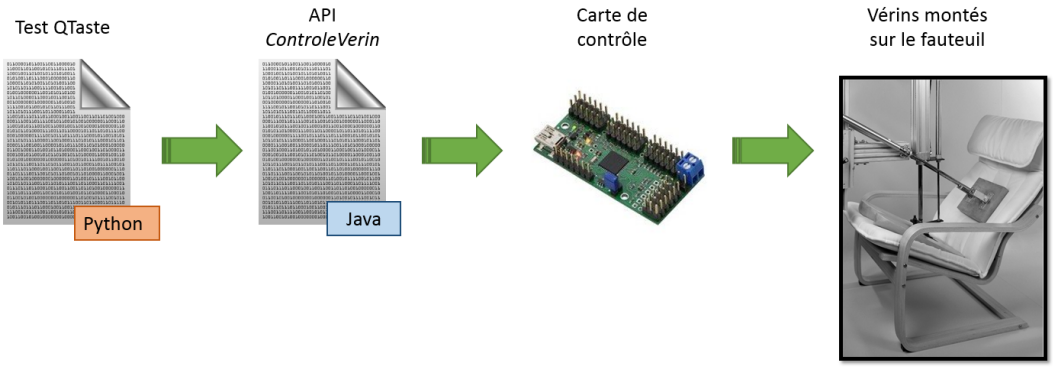


FIGURE 3.3 – Chaîne de tests de robustesse d'un fauteuil

Nous souhaitions utiliser cet exemple pour montrer que QTaste ne se limite pas seulement aux tests logiciels mais que les *Test APIs* ouvrent aussi la porte à des tests mécaniques. Pour illustrer notre concrétisateur de tests abstraits, nous utiliserons la *Test API Selenium* [13] qui est disponible avec QTaste et qui nous permettra de contrôler des navigateurs pour tester des applications web ou des sites web.

3.1.4 Format des tests

Comme mentionné précédemment, QTaste demande aux utilisateurs d'écrire leurs tests en Python. Si nous pouvons utiliser le langage comme nous le souhaitons, il faut néanmoins respecter une certaine structure dans les fichiers de tests pour un bon fonctionnement du système de *log* automatisé. On définit en premier lieu les imports. Ensuite, on écrit toutes les fonctions que l'on souhaite. Enfin, on appelle les différentes méthodes qui effectuerons des actions sur le système à l'aide d'une méthode fournie : *dostep(nomMethodeAAppeler)*. Après, QTaste exécutera ce fichier Python qui interagira avec le système à tester grâce à la *Test API*. Sans rentrer dans les détails de l'implémentation, voici ce à quoi pourrait ressembler un fichier de tests correspondant à la Figure 2.12.

```
#Imports
from qtaste import *


#instances
systemeATester = testAPI.getSelenium(INSTANCE_ID='ReseauSocial')


#Methodes
def OuvrirSite():
    """
    @step       Ouvrir le site
    @expected   Le site est ouvert
    """
    #implemtation

def EntrerIdentifiantsValides():
    """
    @step       Entre des identifiants valide dans la page de loggon
    @expected   Les identifiants sont acceptes
    """
    #implemtation

 def PosterStatutDummy():
    """
    @step       Poste un statut
    @expected   Le statut est poste
    """
    #implemtation

 def Logout():
    """
    @step       Quitte la session courante
    @expected   L'utilisateur n'est plus enregistre
    """
    #implemtation

#Execution
doStep(OuvrirSite)
doStep(EntrerIdentifiantsValides)
doStep(PosterStatutDummy)
doStep(Logout)
```

Code potentiel du cas de test 3 de la Figure 2.12

3.1.5 Données

Dans la solution proposée par QTaste, les données ont été séparées des tests. C'est une excellente chose car du côté du concrétisateur de tests, cela permet de faire de même et de continuer de séparer les deux problèmes.
Au niveau des tests QTaste, on fait un lien vers les données grâce à une méthode spécifique et un identifiant pour la donnée voulue. A côté de cela, on crée un fichier CSV avec, en première ligne, les identifiants, et ensuite on crée autant de lignes que

l'on souhaite de valeurs pour ces différentes données. L'exemple ci-dessous illustre plusieurs valeurs d'identifiants et mots de passe :

```
COMMENT ;BROWSER     ;LOGIN       ;PASSWORD;;
login1  ;*iexplore   ;Andrev      ;12345678;;
login2  ;*iexplore   ;JeanYves    ;DiGurakO2;;
```

données de test

Le test qui utilise ce fichier CSV sera joué autant de fois qu'il y a de ligne de données. Donc si le test Python du point précédent fait référence à une des variables ci-dessus, ce même code sera rejoué deux fois, avec à chaque fois, des valeurs différentes.

3.1.6 Rapports

Pour chaque test joué, QTaste fournit un rapport *HTML* (Figure 3.4) qui reprend essentiellement :

— Le nom du test.

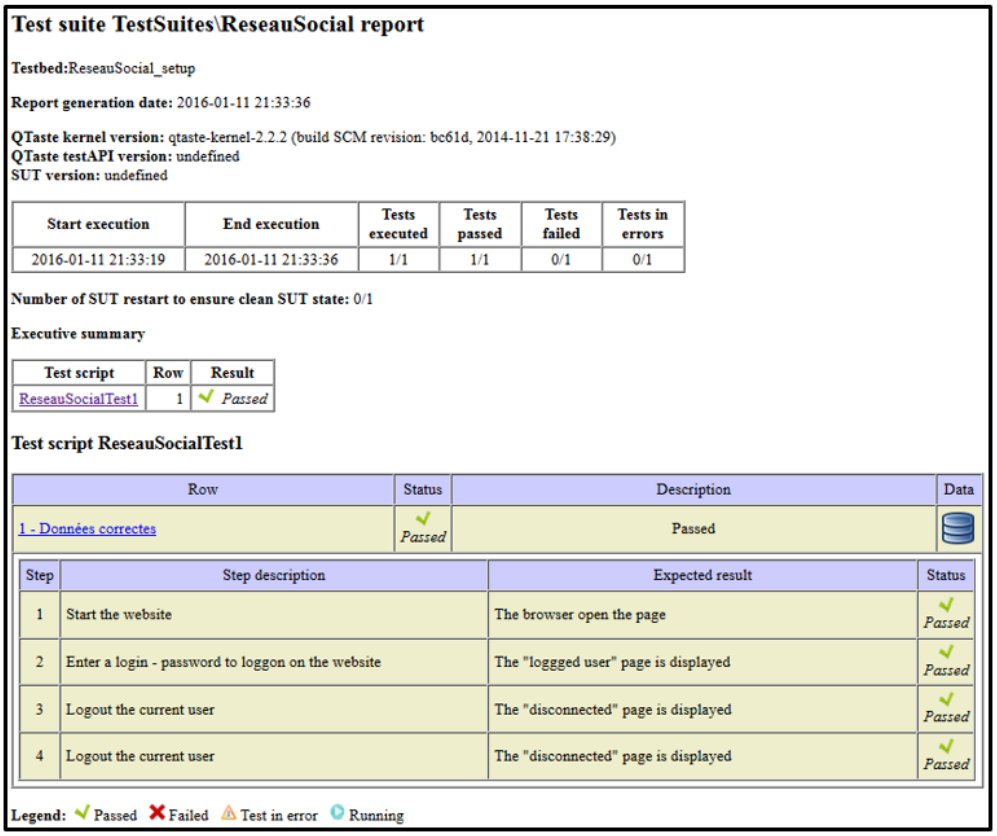
— Le *Testbed* utilisé et son détail. Il s'agit du fichier de mise en place de l'environnement. Ce concept sera expliqué dans le point 3.1.7.

— L'heure du démarrage d'exécution et l'heure de la fin.

— Si le test a échoué, réussi ou est tombé en erreur.

— Les données utilisées pour chaque exécution.

— Pour chaque exécution, les **actions** appelées.



Test suite TestSuites\ReseauSocial report

Testbed: ReseauSocial_setup

Report generation date: 2016-01-11 21:33:36

QTaste kernel version: qtaste-kernel-2.2.2 (build SCM revision: bc61d, 2014-11-21 17:38:29)
QTaste testAPI version: undefined
SUT version: undefined

Start execution	End execution	Tests executed	Tests passed	Tests failed	Tests in errors
2016-01-11 21:33:19	2016-01-11 21:33:36	1/1	1/1	0/1	0/1

Number of SUT restart to ensure clean SUT state: 0/1

Executive summary

Test script	Row	Result
ReseauSocialTest1	1	Passed

Test script ReseauSocialTest1

Row	Status	Description	Data
1 - Données correctes	Passed	Passed	

Step	Step description	Expected result	Status
1	Start the website	The browser open the page	Passed
2	Enter a login - password to loggon on the website	The "loggged user" page is displayed	Passed
3	Logout the current user	The "disconnected" page is displayed	Passed
4	Logout the current user	The "disconnected" page is displayed	Passed

Legend: Passed Failed Test in error Running

FIGURE 3.4 – Un exemple de rapport HTML

3.1.7 Mise en place

Avant de démarrer une série de tests, QTaste propose de démarrer le *TestBed* au préalable. Comme son nom l'indique (littéralement "lit de test"), il s'agit du milieu dans lequel vont se jouer les tests. Ce *TestBed* permet entre autres de faire complètement abstraction de la notion d'environnement lors de l'écriture des tests. On ne doit idéalement voir aucun démarrage ou aucune configuration de processus dans un fichier de test Python directement. Le *TestBed* est composé d'un fichier XML de configuration et d'un "script de contrôle".

Le fichier de configuration contient les chemins vers les *Test APIs*, le chemin vers le script de contrôle, et des informations utiles pour les *Test APIs* ou les hypothétiques processus à démarrer.

Les scripts de contrôle peuvent accéder aux informations du fichier XML grâce à des méthodes proposées par QTaste. Malheureusement toutes les méthodes accessibles au sein des scripts de contrôle ne sont pas documentées mais l'implémentation est disponible puisque c'est un outil *Open Source*. Le but de ces scripts est de pouvoir démarrer des processus nécessaires au déroulement des tests. Si on prend le cas de Selenium qu'on utilise dans nos exemples, il faut démarrer un exécutable Java (*Selenium server*). Les scripts de contrôle ne sont pas les seuls à pouvoir accéder au fichier de configuration du *TestBed* ; les *Test APIs* peuvent le faire aussi. Et de nouveau, les *Test APIs* écrites en Java ont des méthodes pour accéder au TestBed. On peut par exemple définir que pour l'instance de Selenium qui porte l'ID "ReseauSocial", l'URL de démarrage sera "http ://monreseausocial.be".

Les bouts de code ci-dessous proviennent des fichiers “demo” de QTaste [7]. Ils montrent un exemple (adapté à notre réseau social) de *TestBed* et de son script de contrôle :

```
<testbed_configuration>
  <testapi_implementation>
    <import>com.qspin.qtaste.testapi.impl.testexecutor</import>
  </testapi_implementation>
  <control_script>control_selenium.py</control_script>
  <multiple_instances_components>
    <Selenium id="ReseauSocial">
      <host>localhost</host>
      <port>4444</port>
      <url>http://monreseausocial.be</url>
    </Selenium>
  </multiple_instances_components>
</testbed_configuration>
```

configuration du TestBed

```
from controlscript import *

ControlScript([
    JavaProcess("Selenium Server",
                mainClassOrJar="testexecutor/selenium-server-standalone
    -2.41.0.jar",
                checkAfter=10)
])
```

Script de controle

On voit ici que le script de contrôle ne fait qu’un appel pour lancer l’exécutable Java (*Selenium server*) et que les informations propres aux instances de Selenium seront lues dans l’implémentation de cette *Test API* :

```
public SeleniumImpl(String instanceId) {
        super();
        TestBedConfiguration config = TestBedConfiguration.getInstance
    ();
        this.instanceId = instanceId;
        this.host = config.getMIString(instanceId, "Selenium", "host");
        this.port = config.getMIInt(instanceId, "Selenium", "port");
        this.URL = config.getMIString(instanceId, "Selenium", "url");
        logger.debug("Target url : " + this.URL);
}
```

Constructeur de l’API de Selenium [13]

3.1.8 Système de plugins/*AddOn*

En parcourant les sources de QTaste, nous avons observé qu'un système de plugin/*AddOn* (les deux termes sont employés pour définir le même concept) a été mis en place et continue à être maintenu. Ce système n'est cité que d'une manière anecdotique dans la documentation sans en détailler l'usage. En allant voir les différentes branches sur le *GitHub* de QTaste, nous avons découvert une branche qui présentait justement l'implémentation de ceux-ci (Figure 3.5).

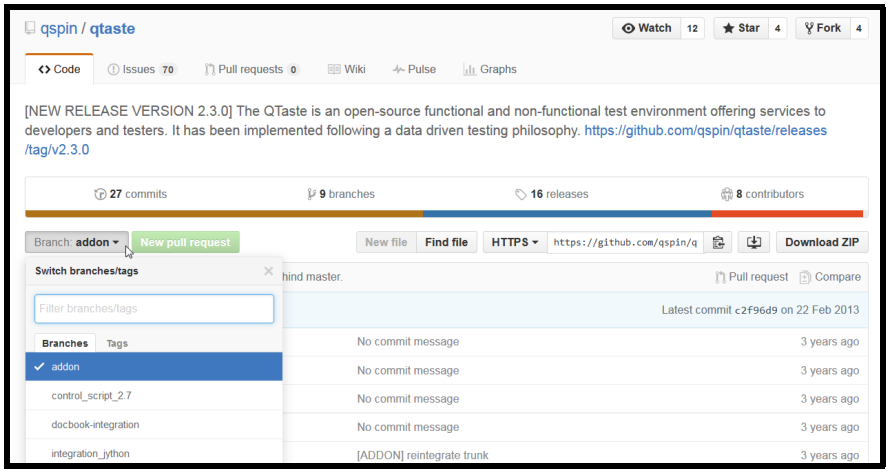


FIGURE 3.5 – La branche qui présente les *AddOns*

Dans les faits, QTaste fournit une zone de l'interface graphique dédiée aux *AddOns*. Tout *AddOn* peut donc profiter de cette zone pour y présenter sa propre IHM (Interface Homme-Machine). Tout *AddOn* doit simplement hériter de la classe "*AddOn*" implémentée dans les sources de QTaste et doit définir les méthodes présentées ci-dessous :

```
import javax.swing.JPanel;
import com.qspin.qtaste.addon.AddOn;
import com.qspin.qtaste.addon.AddOnException;
import com.qspin.qtaste.addon.AddOnMetadata;


public class YourAddon extends AddOn
{

  public YourAddon(AddOnMetadata pMetaData) {
    super(pMetaData);
    //...
  }

  @Override
  public boolean loadAddOn() throws AddOnException {
    //...
```

```
    }

    @Override
    public boolean unloadAddOn() throws AddOnException {
        //...
    }


    @Override
    public boolean hasConfiguration() {
        //return true if the addon has a GUI
    }

    @Override
    public JPanel getConfigurationPane() {
        //build de GUI
    }
}
```

Hériter de la classe *AddOn* de QTaste [4]

3.2 Interfaçage avec VIBeS

Le but de VIBeS est de produire des CTAs qui soient les plus efficaces possible en terme de couverture. C'est-à-dire qu'il va générer un nombre de CTAs le plus petit possible tout en étant complet au niveau de la couverture. L'outil génère un fichier XML qui contient les tests abstraits sous la forme d'une série de transitions qui se succèdent. Il a donc fallu penser à l'enrichir pour que l'on puisse, depuis ce fichier de sortie, créer des tests exécutables. Pour cela nous avons intégré la notion d'assertion liée à un état, entre deux actions.

3.2.1 Enrichissement du modèle

Il faut que la sortie de VIBeS soit en correspondance avec la Figure 2.5. Nous avons donc imaginé une structure extrêmement simple et efficace. Elle pourrait évoluer très facilement. On ne trouvera dans ce fichier de sortie qu'une suite d'actions et d'assertions pour chaque test. Le cas de test de la Figure 2.14 peut donc être représenté de manière complète grâce à l'XML suivant :

```
<test>
  <assert> LoggedOut </assert>
  <action> EntrerIdentifiantsValides </action>
  <assert> VerifierIdentifiantcourant </assert>
  <action> PosterStatutDummy </action>
  <assert> VerifierIdentifiantcourant </assert>
  <action> LikerStatut </action>
  <assert> VerifierIdentifiantcourant </assert>
  <action> NePlusLikerStatut </action>
  <assert> VerifierIdentifiantcourant </assert>
  <action> Logout </action>
  <assert> LoggedOut </assert>
  <action> Exit </action>
</test>
```

Sortie modifiée pour VIBeS

On peut observer que le fichier XML couvre entièrement les détails du test abstrait. Néanmoins, ce fichier ne suffit pas à lui seul pour créer un cas de test concret. Il nous manque le détail d'implémentation des "stimulis" (ou "actions") et "asserts".

3.3 Méthodologie de concrétisation

Précédemment, nous avons analysé très brièvement le système de *mapping* que STALE [22] proposait pour concrétiser du code. C'est ce système qui permet de convertir des instructions abstraites en code concret grâce à des fichiers annexes que l'utilisateur doit fournir. Ces fichiers contiennent, sous une syntaxe très particulière, le code de chaque opération, les variables globales, etc... Nous ne voulions pas suivre cette manière de faire. En effet, faire cela conduirait aux conséquences suivante :

Premièrement, si cette méthodologie nous fournit une méthode cadrée et une manière simplifié pour l'utilisateur de définir les *mappings*, du fait de sa syntaxe simplifiée, elle limite de manière drastique les possibilités qu'offre le langage de base (Python). L'utilisateur ne pourra plus utiliser des librairies externes ou autres. De plus, elle complexifie l'implémentation et la documentation de notre outil puisqu'il faudra définir une syntaxe et une sémantique et créer un parseur pour ces dernières.

Demander à l'utilisateur de fournir du code brut n'est pas réellement une bonne idée non plus car ce serait extrêmement compliqué à gérer au niveau des classes, méthodes, variables globales, etc...C'est pourquoi, nous avons voulu développer une méthodologie qui permettrait à l'utilisateur d'écrire du code Python avec toutes les possibilités que cela implique tout en restant **encadré**.

Cette méthodologie implique une division du problème en plusieurs hiérarchies. Nous avons voulu que le code des "stimulis" et des "*asserts*" soit indépendant de l'implémentation des *Test APIs*. Les différentes hiérarchies identifiées sont les suivantes :

— Le modèle d'interface utilisateur (que l'on appelera aussi "modèle IU") qui définira les types d'accesseurs du système à tester.

— Un *mapping* d'interface utilisateur ("*mapping* IU") qui contiendra les instances des éléments qui permettent l'accès au système à tester.

— Un *mapping* d'opérations qui contiendra l'implémentation des stimulis et des assertions utilisées dans les CTAs.

— Un *mapping* de données qui seront utilisées lors de l'exécution des tests.

3.3.1 Un exemple de *mapping* : Site internet

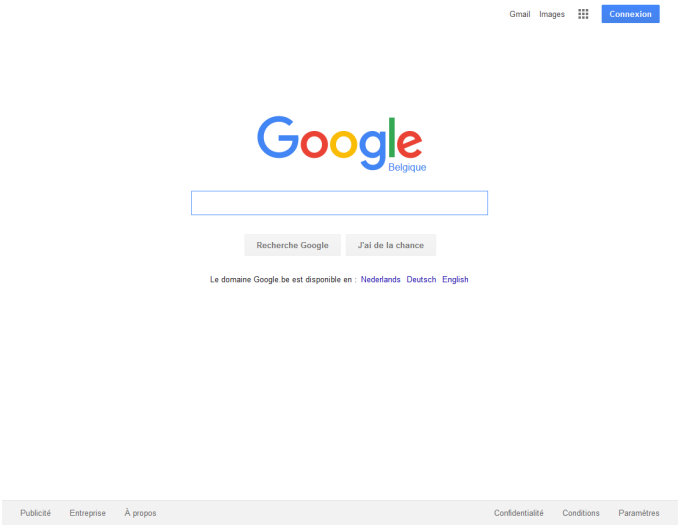


FIGURE 3.6 – La page d'accueil de Google

Pour illustrer cette notion de *mapping*, nous oublions notre réseau social simplifié pour se baser sur un cas réel : Google [5]. Prenons l'exemple de sa page d'accueil présente sur la Figure 3.6. Le modèle d'interface utilisateur demande d'identifier les moyens d'accès au système. Si nous nous arrêtons à cette page, nous pouvons identifier les éléments suivant :

— Des pages Web.
— Des images.
— Des champs texte.
— Des boutons.
— Des labels texte.
— Des liens.

Le premier *mapping* (*mapping* IU) demande d'identifier les instances du modèle IU. Pour cet exemple, en se concentrant sur les éléments principaux, nous avons :

— Une page https ://google.be.
— Une image avec le logo Google.
— Un champ texte de recherche.
— Un bouton "Recherche Google".
— Un bouton "J'ai de la chance".
— Un lien "Nederlands".
— Un lien "Deutsch".
— Un lien "English".

Le *mapping* d'opérations contiendra le code qui utilisera le *mapping* de l'interface utilisateur. Nous y trouverons par exemple du code qui décrit le fait d'entrer du texte dans le champ de recherche et de cliquer sur le bouton "J'ai de la chance".

Enfin, le dernier *mapping* contient les données que l'on voudrait utiliser dans le test. Par exemple, nous pourrions y placer les contenus des recherches "Google" que l'on souhaiterait faire.

3.3.2 Modèle de l'interface utilisateur du système à tester

Nous préciserons ici les différents types d'objets du système auxquels nous devons avoir accès. Nous implémenterons aussi des méthodes génériques qui permettront d'accéder à ces éléments et d'agir dessus (cliquer, entrer texte, naviguer, etc...). Le but de ce modèle est de faire une abstraction par rapport aux méthodes de la *Test API*. Si une *Test API* permet d'accéder au système de manière complète/complexe, le modèle IU doit simplifier cet accès et le rendre invisible aux yeux de la personne qui gère les tests. Ci-dessous, un exemple illustrant une classe du modèle IU d'une interface web, réalisé pour ce travail :

```
#-----------------------------------------
#Web Button
class WebButton:

    def __init__(self, accessMethod, accessValue):
        self.accessMethod = accessMethod
        self.accessValue = accessValue
        self.sut = testAPI.getSelenium(INSTANCE_ID='Google')

    def click(self):
        self.sut.click(self.accessMethod + "=" + self.accessValue)

#-----------------------------------------
```

Un bouton issu du modèle de l'interface utilisateur d'une interface web

3.3.3 *Mapping* de l'inteface utilisateur

Ici, nous définirons les instances des accesseurs définis dans le modèle IU. Comme ce dernier propose des méthodes génériques, lors de la créations de ces instances, nous fournirons ce qui est nécessaire à l'identification du type de moyen d'accès spécifique sur le système. On peut observer dans l'exemple ci-dessous la définition du bouton de *logout* de notre réseau social simplifié :

```
#Definition des instances
btnLogout = modele_web.WebButton("name","logout")
```

Définition des variables d'instance

3.3.4 *Mapping* d'opérations

Dans ce *mapping*, on implémentera toutes les opérations utilisées dans le modèle du système. L'implémentation de ces opérations ne devra utiliser que les variables

définies plus haut et ne devrait pas accéder aux méthodes de la *Test API* directement. L'illustration ci-dessous montre l'implémentation de l'action *Logout()* utilisée dans le modèle de notre réseau social simplifié :

```
def Logout():
    """
    @step       Quitte la session courante
    @expected   L'utilisateur n'est plus enregistre
    """
    variables.boutonLogout.click()
```

Implémentation d'une méthode

3.3.5 *Mapping* de données

Ce dernier *mapping* contiendra simplement les données que nous souhaitons utiliser pour nos tests. Il devra respecter le format attendu par QTaste, à savoir, un simple fichier CSV, comme présenté à la section 3.1.5.

3.3.6 En clair

Cette méthodologie fournit une architecture robuste. Toute modification de *Test API* n'impactera que le modèle IU. Toute modification du modèle IU ne demandera de modifier que le *mapping* IU. Enfin, le fait d'ajouter ou de retirer des variables dans le *mapping* IU n'impacte que le *mapping* d'opérations. La Figure 3.7 présente cette méthodologie de manière graphique.

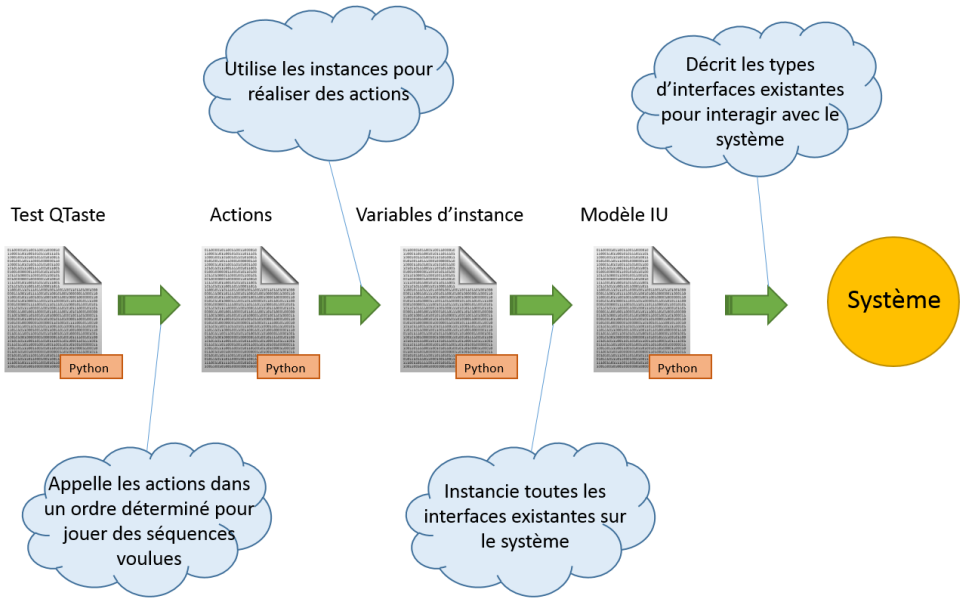


FIGURE 3.7 – Architecture des fichiers de test

3.3.7 Exécution des tests

Si cette méthodologie est suivie, aucun *réel* algorithme de concrétisation n'est nécessaire pour rendre les tests exécutables. En effet, le rôle de notre concrétisateur sera d'ajouter les références entre les fichiers pour que les appels puissent se faire entre le *mapping* d'opérations, le *mapping* IU et le modèle IU. L'outil aura donc un rôle de "formatage" de fichiers Python plus que de "génération".

Concrètement, le fichier "*mapping* opérations" fera un import du fichier "*mapping* IU" qui fera à son tour un import du/des modèle(s) d'interface utilisateur.

3.4 Concrétisation de la méthodologie

Nous avons présenté dans la section précédente la manière dont nous souhaitons transformer des CTAs en tests exécutables. Cependant, nous n'avons pas défini le moyen que nous allons utiliser pour l'implémenter. Nous savons ici qu'il s'agit d'un outil qui se place entre VIBeS et QTaste et nous savons aussi que l'utilisateur doit fournir des *mappings* en Python pour que la concrétisation des tests soit fonctionnelle.
Cet outil devra générer des tests concrets et fonctionner de manière transparente avec QTaste.

Après analyse, trois options s'offrent à nous :

— Modifier les sources de QTaste

— Créer un outil indépendant de QTaste qu'il faut utiliser en amont de celui-ci

— Créer un *AddOn* QTaste

La première option n'est tout simplement pas envisageable. En effet, la société QSpin passe du temps à valider son outil QTaste pour que des entreprises qui travaillent dans des milieux critiques puissent l'utiliser. Modifier ses sources nous enlève cette "certification" et nous diminuons le public visé par notre outil.
La deuxième possibilité présente deux gros désavantages. Le premier est qu'il s'agit d'un outil de plus à utiliser par le testeur, ce qui est déjà une contrainte en soi. De plus, pour écrire les fichiers de tests au bon endroit, l'utilisateur devra mentionner où se trouvent ceux-ci. Enfin, nous n'aurions pas directement accès aux fonctions fournies dans l'implémentation ouverte de QTaste.
La troisième et dernière possibilité est tout simplement la bonne. En effet, les *AddOns* sont des modules *plug&play* qui donnent un accès à toute l'implémentation de QTaste. C'est cette option qui sera retenue pour implémenter notre outil.

Chapitre 4

AbsCon, un *AddOn* QTaste

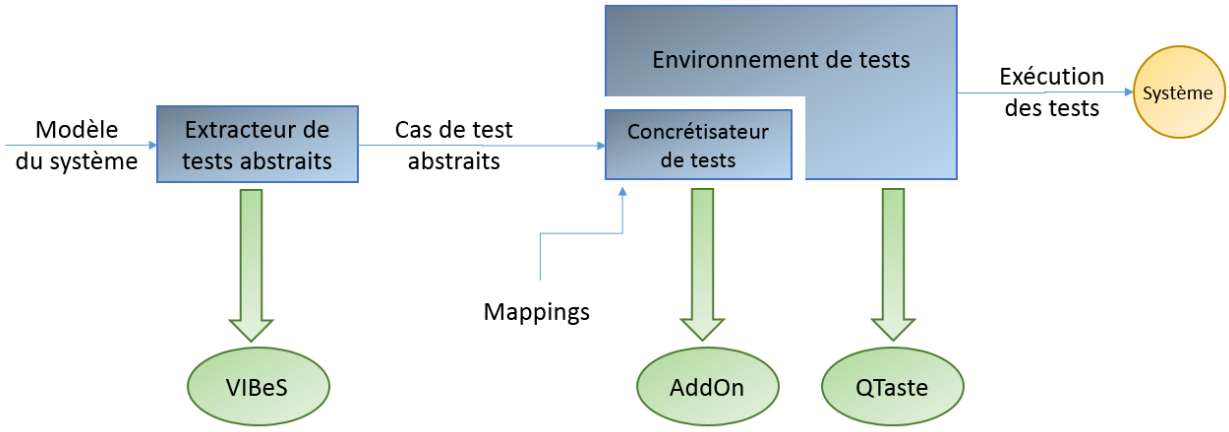


FIGURE 4.1 – Notre chaîne de tests

Nous avons décidé d'utiliser QTaste pour l'exécution de tests puisqu'il s'agit d'un outil ouvert et évolutif. Nous savons que notre concrétisateur sera un *AddOn* QTaste qui sera directement intégré dans son architecture. Enfin, nous savons désormais qu'il y aura des fichiers de *mapping* représentant les différents niveaux d'abstraction définis à la section 3.3.

4.1 AbsCon

En se plaçant dans la peau d'un testeur, nous nous sommes demandé quels seraient les moyens les plus agréables pour rentrer les différentes informations attendues de nous (les *mappings*). Nous avons donc commencé par dessiner les *mockups* des IHM de l'*AddOn*. La zone de l'interface graphique de QTaste à laquelle nous avons accès via les *AddOns* a été entourée d'une large ligne noire.

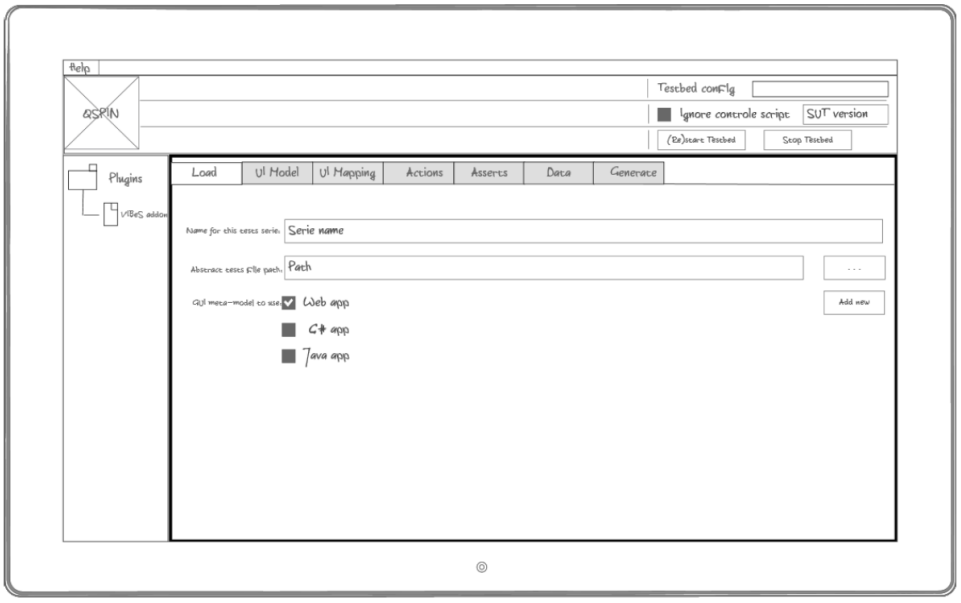


FIGURE 4.2 – Ecran 1 : la fenêtre d'accueil

La Figure 4.2 présente l'écran d'accueil. Sur cet écran, l'utilisateur définit un nom pour sa série de tests. Il sera ausi invité à *uploader* le fichier XML qui définit les cas de tests abstraits (sortie de VIBeS). Enfin, il cochera les modèles IU qu'il souhaite utiliser pour tester son système. Cette liste de modèles doit être raffraîchie automatiquement avec les modèles qui se trouvent dans le répertoire de QTaste "/TestSuites/UIModels".

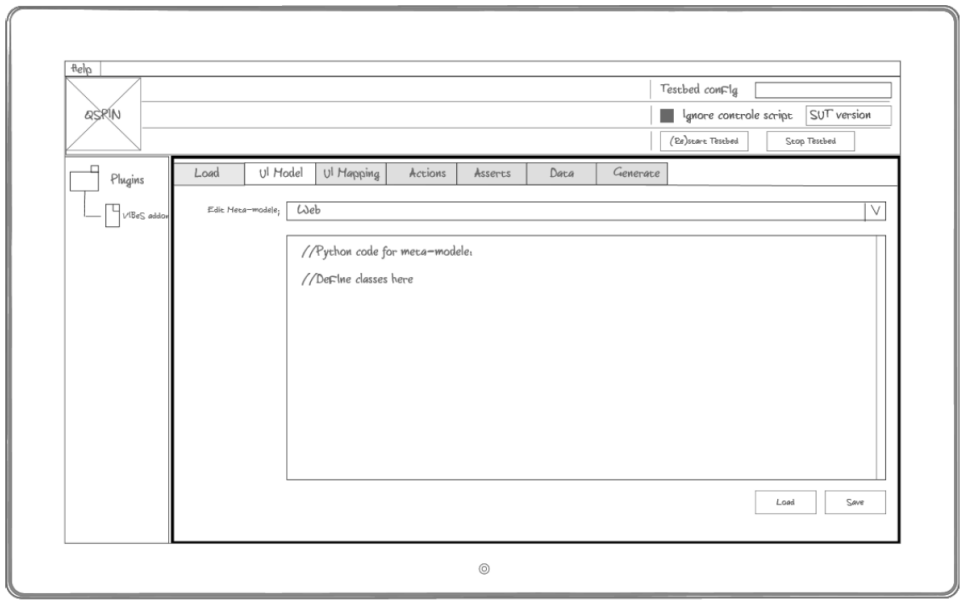


FIGURE 4.3 – Ecran 2 : la fenêtre d'édition des modèles IU

Dans le deuxième écran illustré sur la Figure 4.3, l'utilisateur peut accéder à l'implémentation des modèles d'interface utilisateur sélectionnés. Le but principal est d'avoir une vue sur le code mais aussi de le modifier. Normalement, un modèle IU bien rédigé ne devrait pas être modifié et il devrait convenir pour tous les tests qui doivent s'exécuter sur le système visé. La procédure exacte demandée pour la définition de ces modèles IU sera détaillée dans la section 4.2. Un bouton devra permettre de recharger le contenu du fichier et un autre de le sauver.

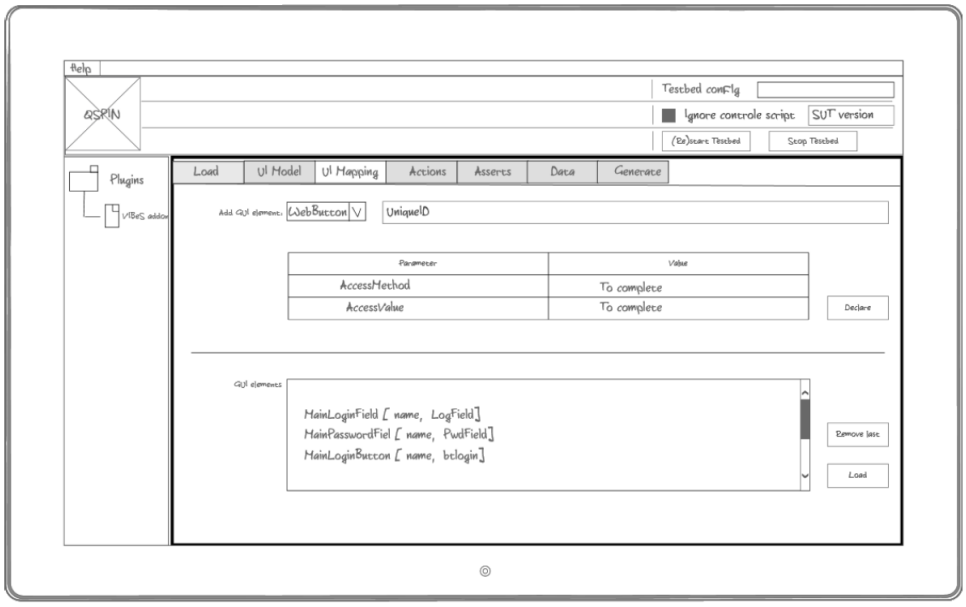


FIGURE 4.4 – Ecran 3 : la fenêtre de définition du *mapping* de l'IHM à tester

L'écran de la Figure 4.4 illustre de manière parfaite l'intéret de l'interface graphique proposée par l'*AddOn* pour aider l'utilisateur à fournir les informations que nous attendons de lui. L'IHM imaginée ici a été pensée d'une telle manière que l'utilisateur ne doive pas se soucier du langage final (Python).

Un menu déroulant propose les accesseurs du système à instancier. Cette liste sera remplie grâce aux classes des modèles IU sélectionnés. L'utilisateur peut ensuite définir un nom pour sa variable de *mapping* et, enfin, fournir une valeur aux paramètres qui aideront à identifier l'accesseur. Ces paramètres apparaissent dynamiquement dans le tableau juste en dessous.
Dans le cadre qui compose la deuxième partie de la fenêtre, un résumé des variables définies apparaît. Un bouton permet de supprimer la dernière entrée et un autre permet de télécharger un fichier Python qui contient des *mappings* IU et de les intégrer dans la fenêtre.

FIGURE 4.5 – Ecran 4 : écrire le code python des actions abstraites

Dans la fenêtre de la Figure 4.5, une liste déroulante est automatiquement peuplée lorsque l'utilisateur télécharge le fichier XML de CTAs dans l'écran de la Figure 4.2. Elle contient toutes les actions utilisées dans les différents tests abstraits. On demandera ici à l'utilisateur d'écrire le contenu des méthodes Python. Il devra uniquement se soucier du contenu et non de la structure entre les différents fichiers. Un bouton "Load" permettra de sélectionner un fichier Python et de parser son contenu. Si des méthodes ayant un nom identique aux actions sont trouvées, leur contenu sera importé.

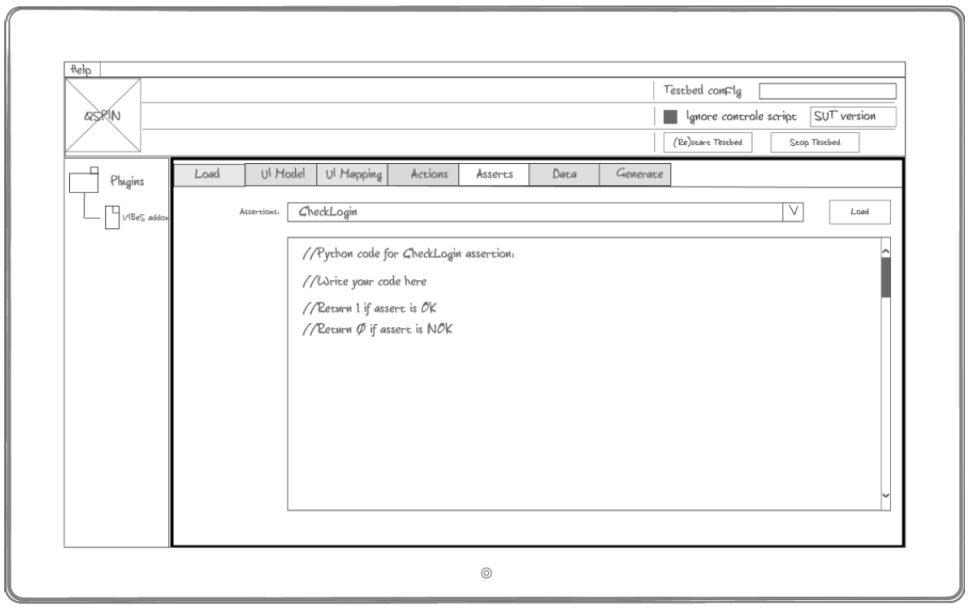


FIGURE 4.6 – Ecran 5 : écrire le code Python des assertions abstraites

L'écran de la Figure 4.6 suit le même fonctionnement que l'écran précédent. La seule différence est dans le contenu du code ; une assertion doit retourner un '1' ou un '0'.

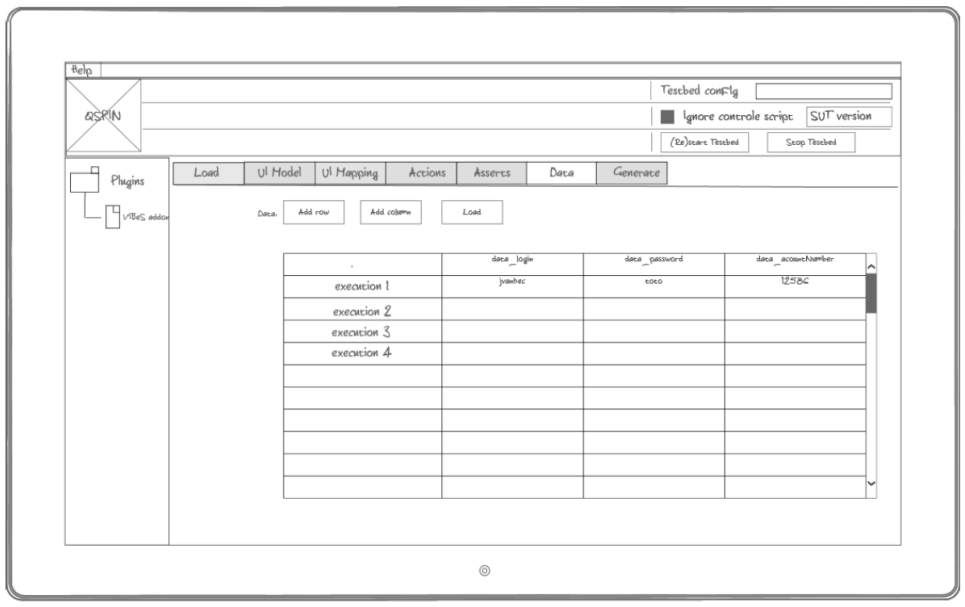


FIGURE 4.7 – Ecran 6 : définition des données de test

La fenêtre de la Figure 4.7 a été pensée pour simplifier la définition des données de test. Initialement au format CSV, les données sont présentées dans un tableau modifiable. Un bouton permettra d'importer un fichier CSV existant dans le tableau.

FIGURE 4.8 – Ecran 7 : création des tests concrets

Enfin, la fenêtre de la Figure 4.8 propose à l'utilisateur de finaliser le processus de concrétisation à l'aide d'un simple bouton.

4.2 *Mappings*

Il est demandé à l'utilisateur de fournir différents fichiers de *mapping*. Il pourra les entrer dans l'interface de l'*AddOn* prévue à cet effet. Les *mappings* qu'on attend de lui doivent respecter certaines structures/restrictions qui sont détaillées ci-dessous.

4.2.1 Modèle d'interface utilisateur

Ce fichier doit contenir le modèle IU du système que l'on souhaite tester. On doit définir une classe Python par type d'objet qui permet l'accès au système (un accesseur). Si notre interface n'est composée que de champs de texte et de boutons, l'utilisateur devra décrire deux classes : *Bouton* et *ChampsTexte*.

Le contenu des classes doit respecter deux contraintes qui se situent toutes deux au niveau du constructeur. Premièrement, tous les paramètres qui seront nécessaires à identifier l'accesseur (via la *Test API*) doivent être reçus par le constructeur de la classe. Deuxièmement, comme l'instance de la classe devra accéder au système, le constructeur doit définir une variable d'instance de la *Test API* qu'on utilise. Ce dernier point implique une forte liaison entre la *Test API* et le modèle IU. Cependant, comme nous l'avons vu à la section 3.1.7, nous pouvons définir plusieurs instances de la *Test API* via des *IDs*. Il ne suffira qu'à définir une instance de la *Test API* ("Selenium" dans notre exemple) "modeleIU_test" dans le testBed et dans notre modèle IU. Ensuite, dans chaque constructeur, nous pourrons récupérer cette instance grâce à la ligne suivante :

```
self.api = testAPI.getSelenium(INSTANCE_ID='modeleIU_test')
```

Récupérer l'instance de la testAPI liée au modèle IU "modeleIU_test"

Une fois ces deux contraintes respectées, l'utilisateur peut définir les méthodes qu'il souhaite dans les classes *Bouton* et *ChampsTexte*. Ces méthodes peuvent contenir des paramètres, mais idéalement toutes les variables dont elles auraient besoin pour identifier l'élément dans l'interface doivent, elles, être fournies via le constructeur de la classe et dans les paramètres de la méthode.

4.2.2 *Mapping* de l'interface utilisateur

L'IHM de AbsCon aide l'utilisateur à définir les instances des accesseurs définis dans le(s) modèle(s) d'interface utilisateur. Il suffit de choisir un type d'accesseur, de lui donner un identifiant unique, et de compléter les informations demandées pour pouvoir identifier l'élément.

4.2.3 *Mapping* des opérations

Par "opérations", nous entendons *actions* et *asserts* (pour "assertions"). Cette étape n'a pas de restriction connue à l'heure actuelle. L'utilisateur peut entrer le code qu'il souhaite et, de plus, le langage Python permet d'effectuer des imports

de librairie à l'intérieur d'une fonction. Les *asserts* présentent une différence avec les *actions* : elles doivent impérativement retourner un '1' si la vérification est un succès et '0' sinon.
L'architecture choisie est telle que le code ne devrait pas faire appel aux méthodes de la *TestAPI* directement mais devrait plutôt agir sur les variables de *mapping* définies. Pour accéder à une variable, il suffit de mettre son nom. Par contre, pour accéder aux données définies, il faut faire appel à la méthode suivante :

```
maVariable = testData.getValue(''Nom de la donnee'')
```

Récupérer une donnée depuis une opération ou tout autre script Python

4.2.4 *Mapping* des données

Les données sont simples à définir. Il suffit d'ajouter une donnée via l'IHM de l'*AddOn* AbsCon en lui spécifiant un nom. Ensuite, on peut définir toutes les valeurs possibles.

4.3 Architecture

Le langage de programmation Java a été imposé par QTaste. Nous avons défini une structure complète de classes pour pouvoir implémenter de manière fiable notre *AddOn* AbsCon. Cette structure sera expliquée dans les points suivants.

4.3.1 Diagrammes de classes

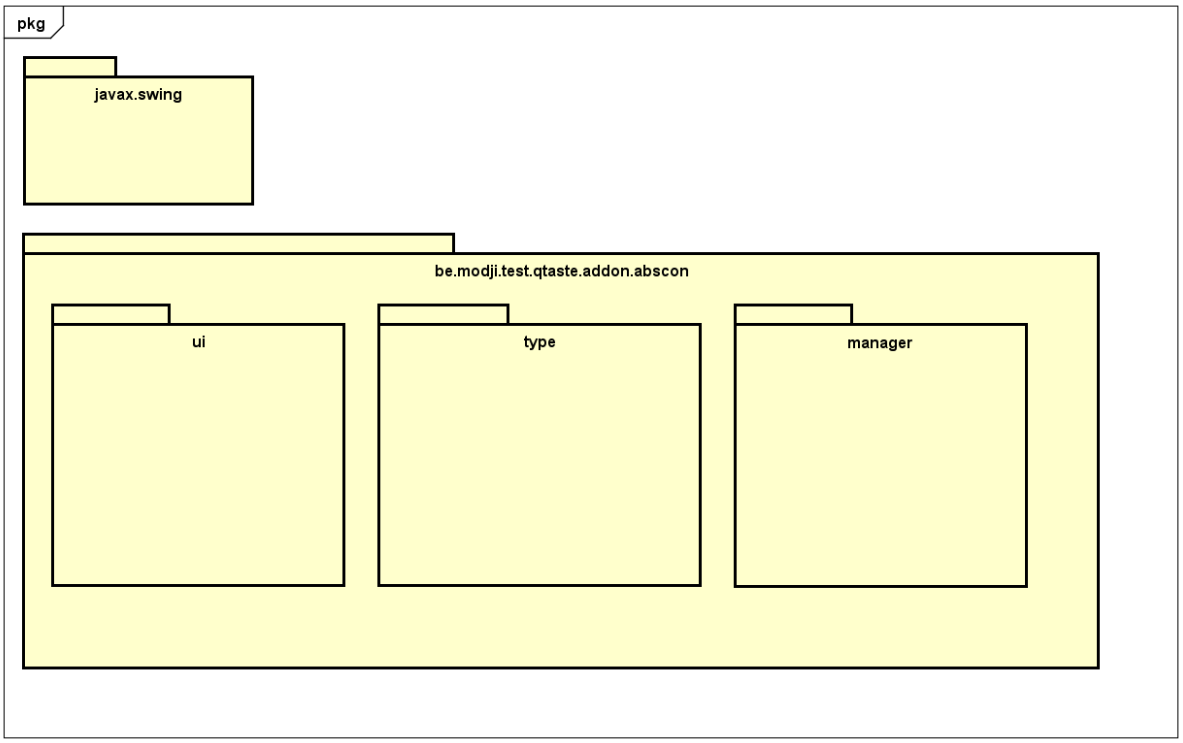


FIGURE 4.9 – Les packages Java utilisés dans l'*AddOn*

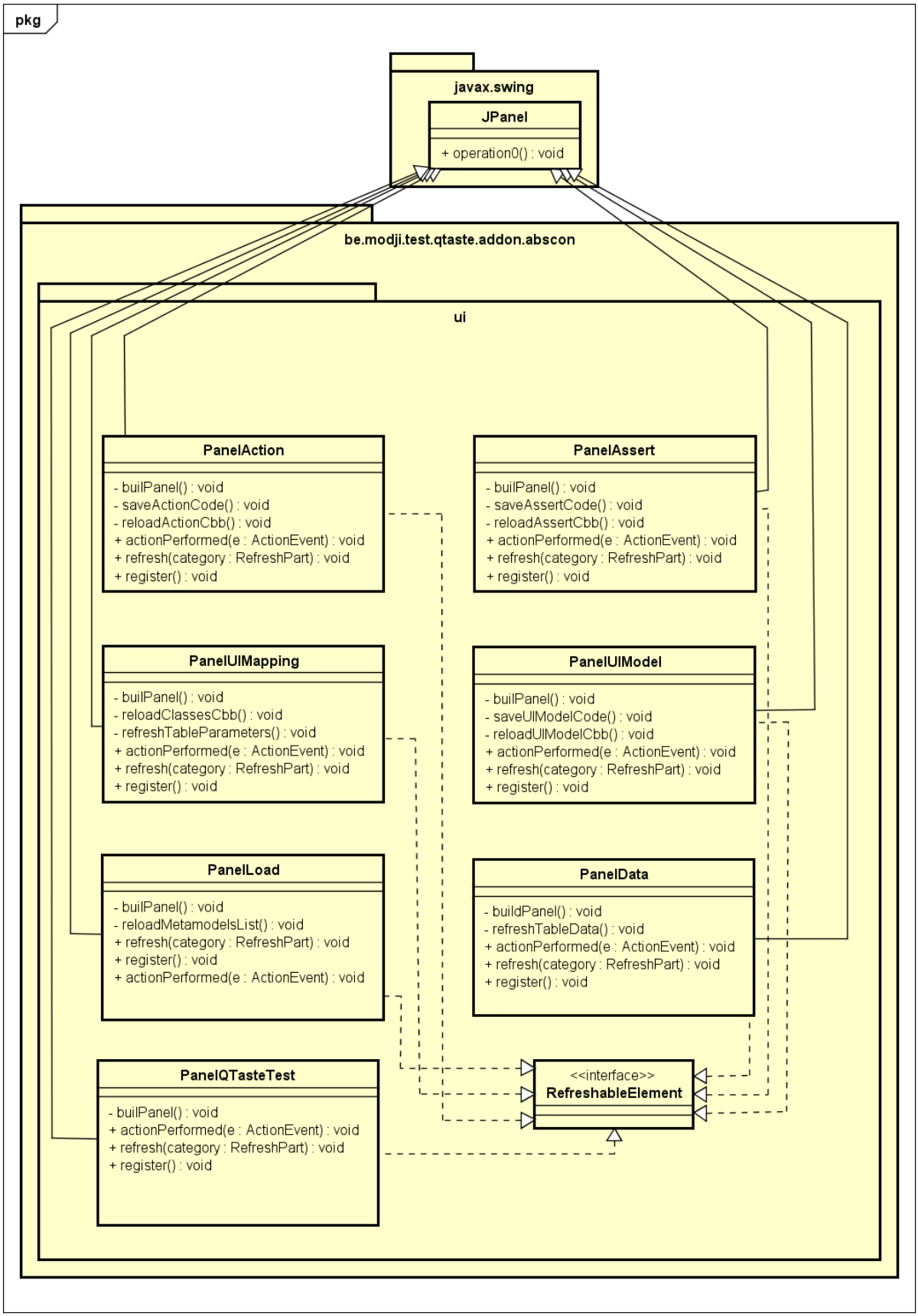


FIGURE 4.10 – Diagramme de classe du package “UI” (pour User Interface)

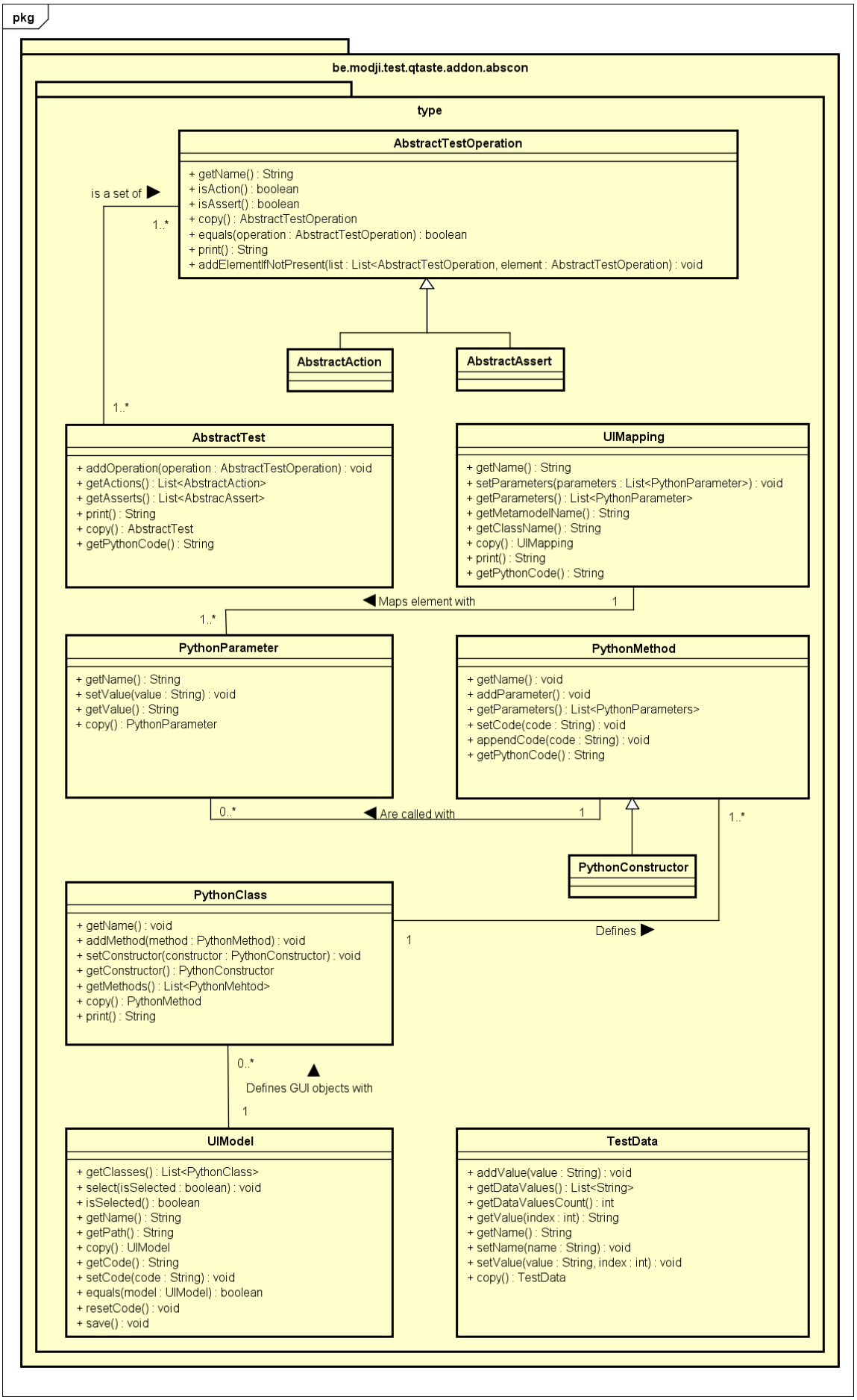


FIGURE 4.11 – Diagramme de classe du package “type”

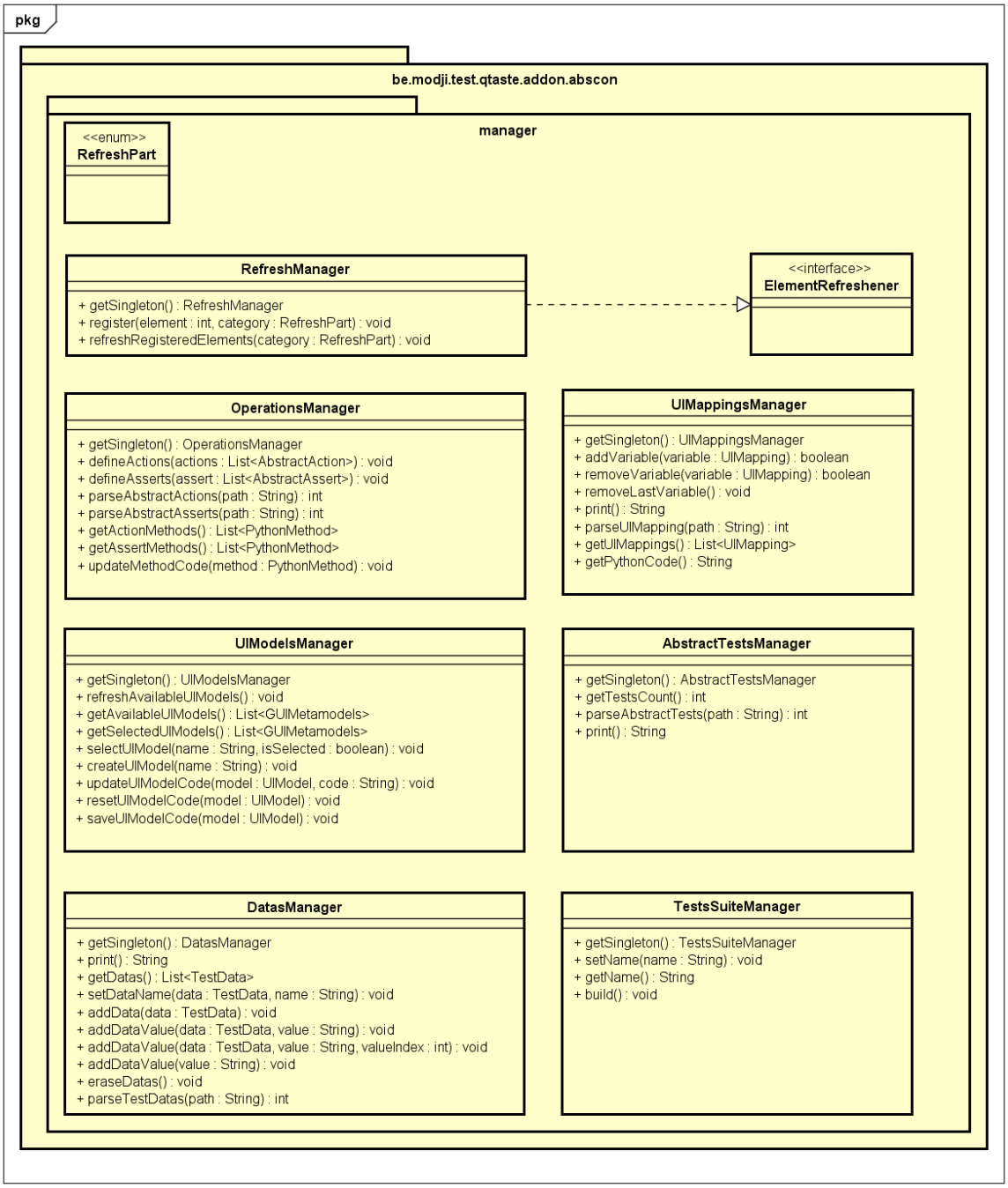


FIGURE 4.12 – Diagramme de classe du package “Managers”

4.3.2 Spécification des types complexes

Pour toute personne souhaitant utiliser le code de l'*AddOn*, nous avons jugé utile de spécifier les différents types complexes représentés dans la Figure 4.11.

AbstractTestOperation : représente une opération abstraite utilisée dans un test abstrait.

- **operationName :** type String représentant le nom de l'opération.

AbstractAction : sous-type de AbstractTestOperation et représente une opération de type "action".

AbstractAssert : sous-type de AbstractTestOperation et représente une opération de type "assert".

AbstractTest : représente un test abstrait issu du fichier de tests abstraits.

- **operations :** type List<AbstractTestOperation> représentant les opérations qui composent le test.

PythonParameter : représente un paramètre d'une méthode Python.

- **parameterName :** type String représentant le nom du paramètre.
- **value :** type String représentant la valeur du paramètre.

PythonMethod : représente les caractéristiques d'une méthode Python.

- **code :** type String contenant le code de la méthode.
- **methodName :** type String représentant le nom de la méthode.
- **parameters :** type List<PythonParameter> représentant les paramètres de la méthode.

PythonConstructor : sous-type de PythonMethod et représente un constructeur.

PythonClass : représente les caractéristiques d'une classe Python.

- **className :** type String représentant le nom de la classe.
- **methods :** type List<PythonMethod> représentant les méthodes implémentées dans la classe.
- **constructor :** type PythonConstructor représentant le constructeur de la classe.

UIModel : représente le modèle IU de l'interface à tester.

- **filePath :** type String représentant le chemin d'accès au fichier.
- **fileContent :** type String représentant le contenu du fichier.
- **modelName :** type String représentant le nom du modèle.
- **newcode :** type String contenant le code modifié du modèle.

UIMapping : représente un élément concret de l'interface à tester.

- **name :** type String représentant le nom de la variable de *mapping*.
- **uiModelName :** type String représentant le nom du modèle IU utilisé.
- **uiModelClass :** type String représentant le nom de la classe à instancier.
- **parameters :** type List<PythonParameter> représentant les paramètres du constructeur à utiliser lors de l'instanciation de la classe.

TestData : représente une donnée à utiliser lors de l'exécution des tests.

- **dataName :** type String représentant le nom de la donnée.
- **dataValues :** type List<String> représentant les valeurs que peut prendre cette donnée.

4.3.3 Spécification des classes “managers”

Les “managers” sont des singletons qui permettent aux fenêtres de l'*AddOn* d'avoir accès à un contexte commun et global. Toute modification du contexte doit être faite via ces managers. Les managers ne peuvent pas renvoyer des données qui, si elles sont modifiées, pourraient altérer le contexte global. Ainsi, les managers ne peuvent renvoyer que des copies de données.

AbstractTestsManager : gère les tests abstraits utilisés dans l'*AddOn*. Il est aussi utilisé pour générer le code Python du fichier de test principal.

DatasManager : gère les données qui seront utilisées pour l'exécution des tests. Il permet aussi de générer le fichier CSV final.

UIMappingsManager : gère les différents *mappings* de l'interface à tester ; il s'occupera de générer le code lié à ces derniers.

UIModelsManager : gère l'affichage des modèles IU et la sauvegarde du code si nécessaire.

OperationsManager : gère les opérations concrètes à générer ; il s'occupera de générer tout le code lié aux opérations.

RefreshManager : gère le rafraîchissement des IHM de l'*AddOn* qui se sont enregistrées pour cet effet.

TestsSuiteManager : gère l'ordonnancement et la génération des tests concrets.

4.4 Structure des fichiers

Pour que l'utilisateur puisse utiliser l'*AddOn*, cette section explique la structure des fichiers et répertoires dans laquelle il doit écrire et lire des fichiers.
Tout d'abord, dans la Figure 4.13, la racine du répertoire d'installation de QTaste :

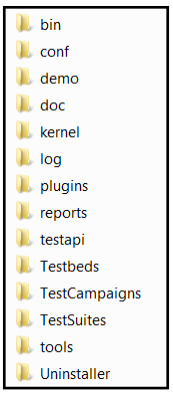


FIGURE 4.13 – Racine du répertoire QTaste

Le répertoire (Figure 4.14) où seront écrits les tests exécutables est le dossier “TestSuites” ; c'est dans ce dossier aussi que l'on pourra trouver :

— Le répertoire qui contiendra les modèles IU.

— Le sous-dossier “pythonLib” ; qui peut être utilisé par l'utilisateur pour y déposer les librairies dont il aurait besoin.

— Les séries de tests créées à partir de l'*AddOn*.

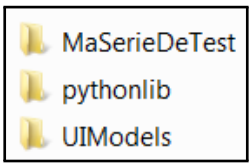


FIGURE 4.14 – Racine du répertoire TestSuites

Dans le répertoire correspondant à une série de tests créée par l'*AddOn* (Figure 4.15), nous pouvons retrouver à nouveau un répertoire qui contient les librairies générées et ensuite, un répertoire par test à exécuter. C'est également dans ce répertoire que nous pourrons trouver le fichier “configuration.ctar”, qui est le fichier généré par l'*AddOn* lors de la génération et qui permet d'importer directement toutes les informations liées à la concrétisation de la série abstraite.

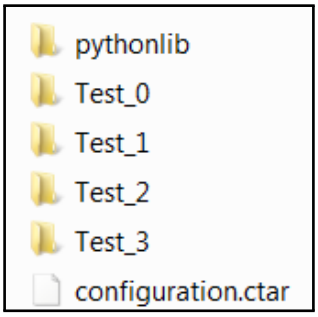


FIGURE 4.15 – Racine du répertoire d'une série de tests générée

La Figure 4.16 montre les librairies générées par l'*AddOn*. Tout d'abord, il y a les modèles IU qui sont simplements copiés et, ensuite, les fichiers de *mappings* d'interface utilisateur et d'opérations.

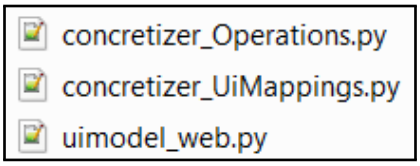


FIGURE 4.16 – Répertoire contenant les librairies d'une série de tests générée

Finalement, sur la Figure 4.17, on peut voir les deux fichiers qui composent un test. Le fichier qui appelle les opérations et, ensuite, le fichier qui contient les données à utiliser pour les tests.

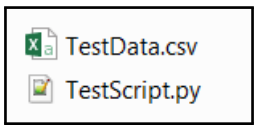


FIGURE 4.17 – Fichiers utilisés pour exécuter le test

4.5 Résultats

Tout en respectant la charte graphique de QTaste, nous avons réussi à respecter le design des *mockups* définis au préalable. Si certaines fonctionnalités semblent parfois manquantes, l'*AddOn* fonctionne de manière fiable et efficace. Il pourra sans aucun doute évoluer dans le futur pour répondre à ces petits manquements pratiques.
Deux ajouts par rapport aux mockups ont tout de même été jugés nécessaires et ont donc été insérés dans la première version de l'*AddOn*. Le premier est un nouvel écran sur lequel on peut visualiser les CTAs chargés. En effet, sans cet écran, nous n'avions aucune vue sur les tests abstraits sélectionnés. Ensuite, l'ajout majeur est la notion de projet. Lorsque des tests sont générés grâce à l'*AddOn*, un fichier ".ctar" est créé. Ce fichier contient toutes les informations concernant les tests concrétisés. Dans la barre de menu de QTaste, l'*AddOn* ajoute un menu qui permet d'importer un projet existant via ce fichier. Une fois un fichier *.ctar* choisi, toutes les informations seront transposées dans les écrans de l'*AddOn* AbsCon.

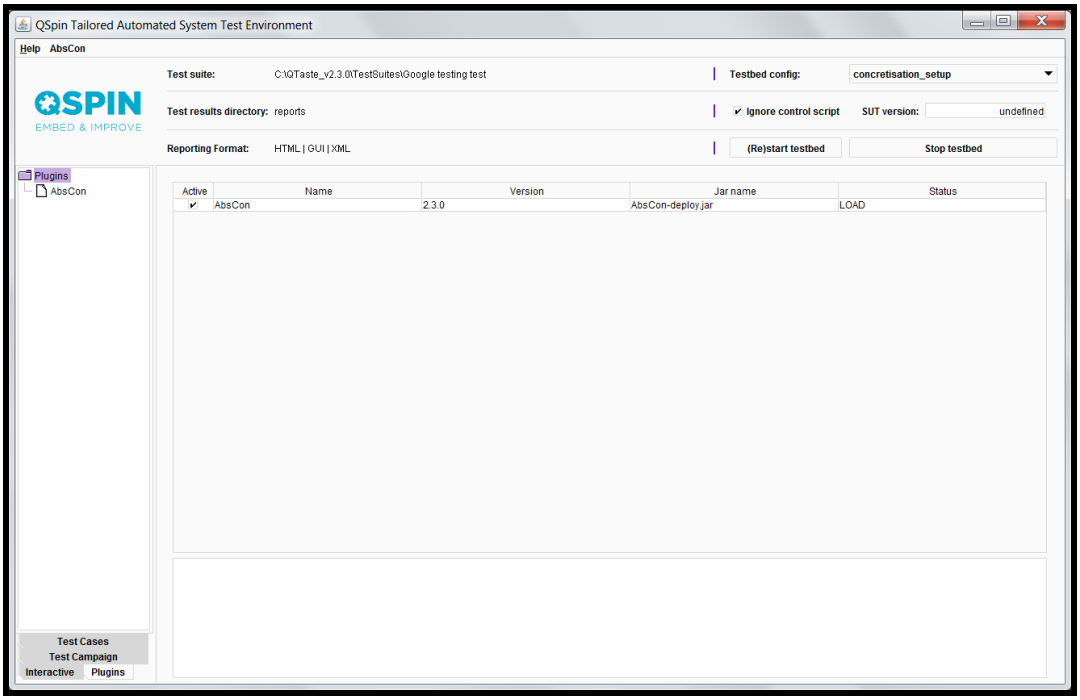


FIGURE 4.18 – Capture d'écran de AbsCon : charger l'*AddOn* dans l'interface de QTaste

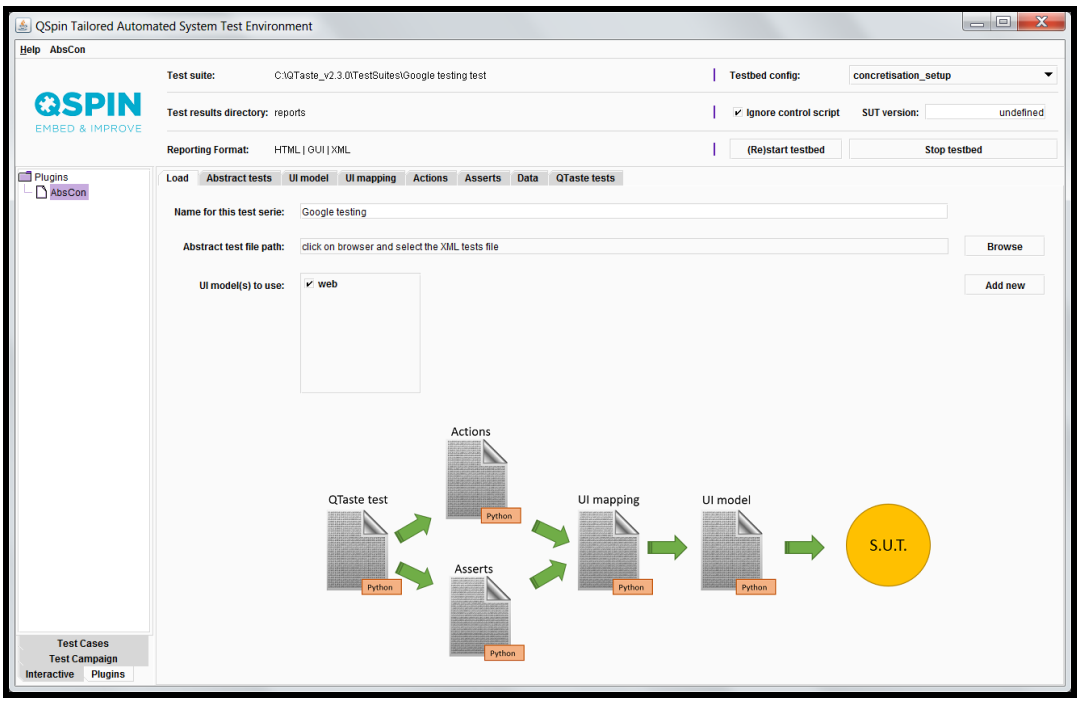


FIGURE 4.19 – Capture d'écran de AbsCon : visualiser l'interface de l'*AddOn* dans l'interface de QTaste

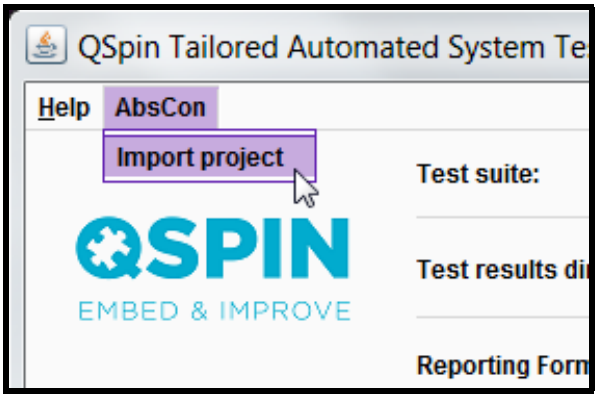


FIGURE 4.20 – Capture d'écran de AbsCon : charger un projet de concrétisation réalisé précédemment

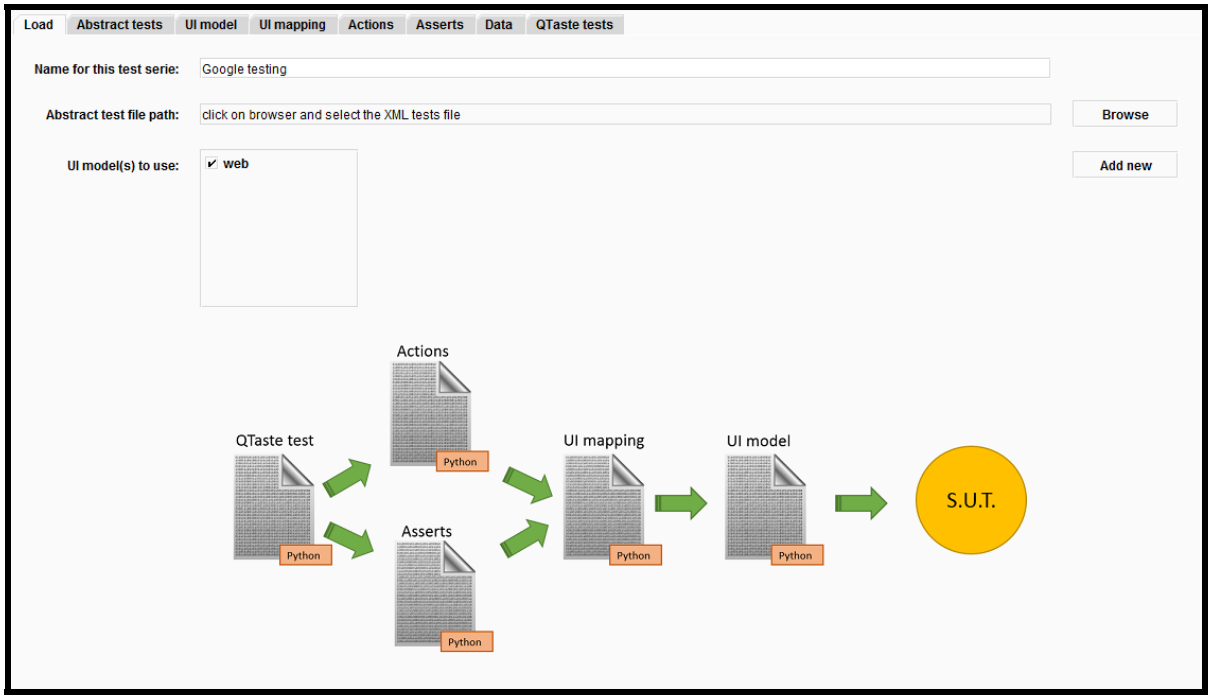


FIGURE 4.21 – Capture d'écran de AbsCon : charger les tests abstraits et sélectionner les modèles d'interface utilisateur

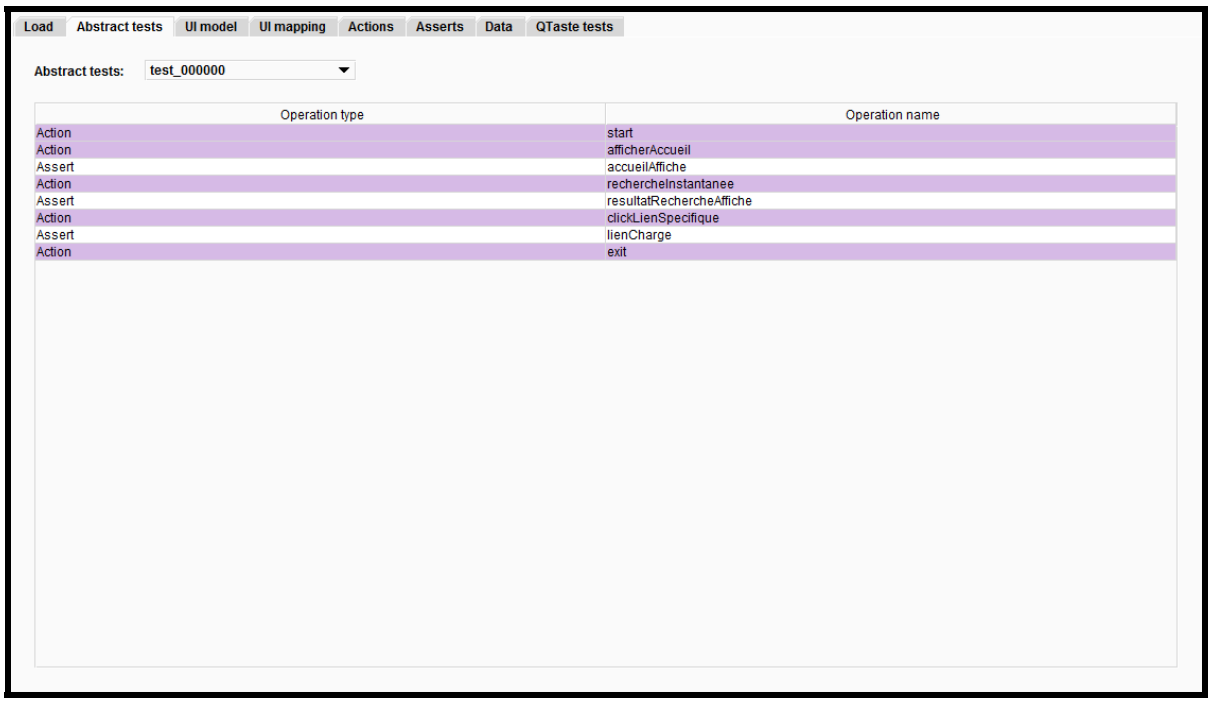


FIGURE 4.22 – Capture d'écran de AbsCon : visualiser les CTAs chargés

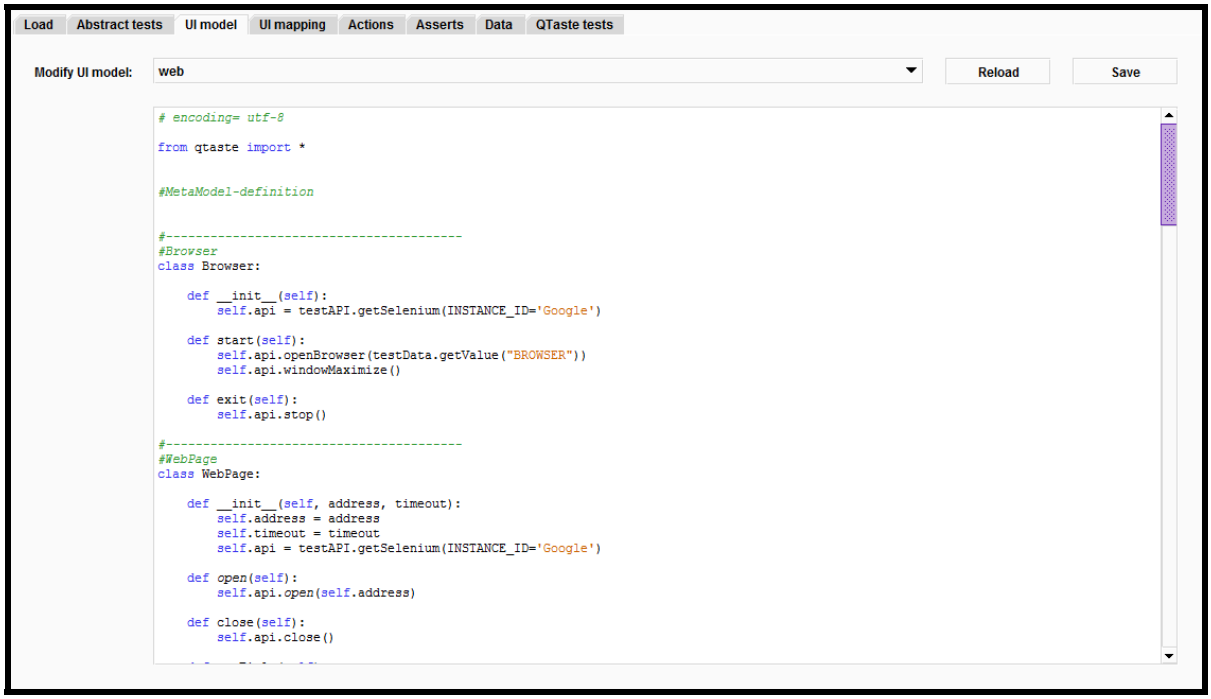


FIGURE 4.23 – Capture d'écran de AbsCon : visualiser ou modifier les modèles IU

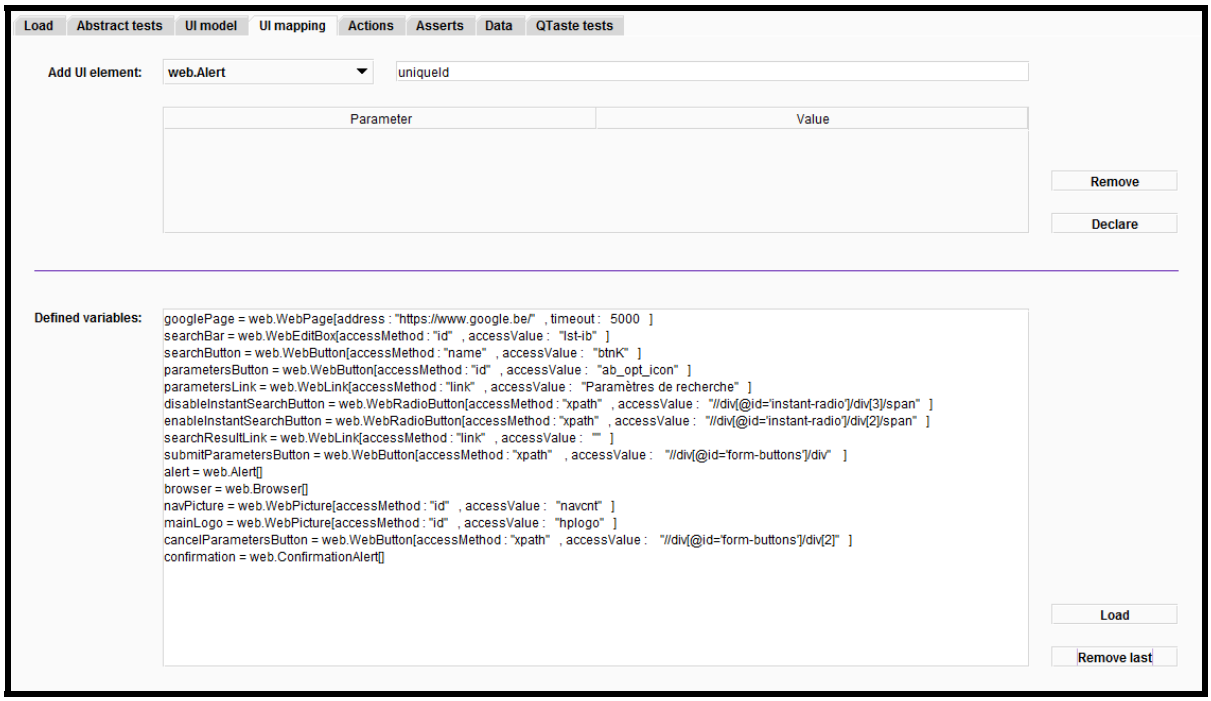


FIGURE 4.24 – Capture d'écran de AbsCon : définir le *mapping* de l'interface à tester

FIGURE 4.25 – Capture d'écran de AbsCon : écrire le code des actions

FIGURE 4.26 – Capture d'écran de AbsCon : écrire le code des assertions

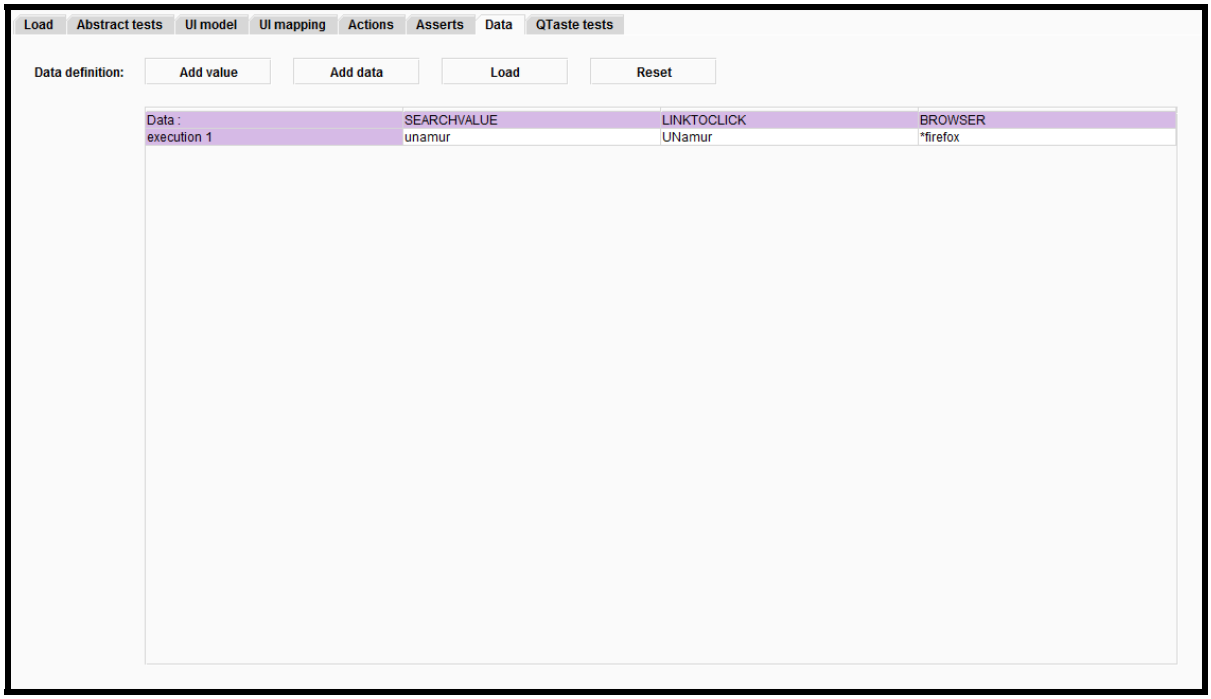


FIGURE 4.27 – Capture d'écran de AbsCon : définir les données à utiliser dans les tests

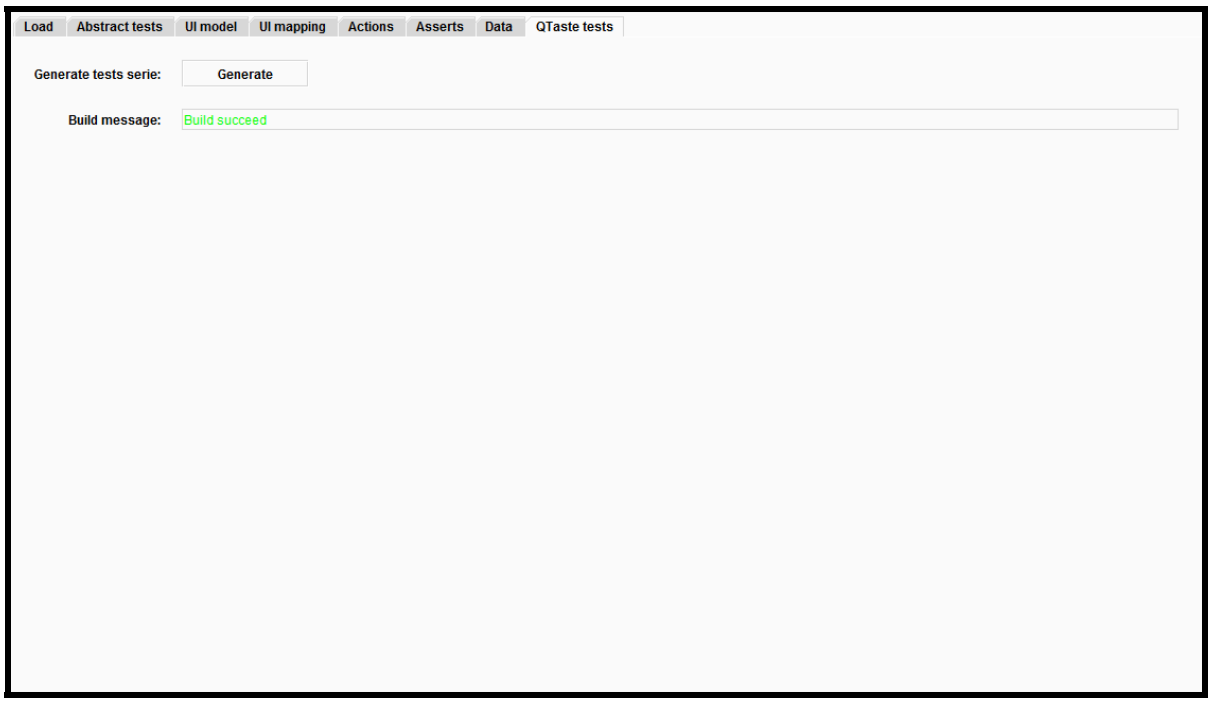


FIGURE 4.28 – Capture d'écran de AbsCon : lancer la compilation des tests

4.6 Cas d'étude : "Google"

Nous montrons dans cette section un exemple concret d'utilisation de l'*AddOn* AbsCon où nous allons tester une partie du site Google [5]. Nous nous bornerons à trois pages :

— la page d'accueil

— la page où les résultats de recherche sont affichés

— une page des paramètres de recherche

4.6.1 Le système à tester

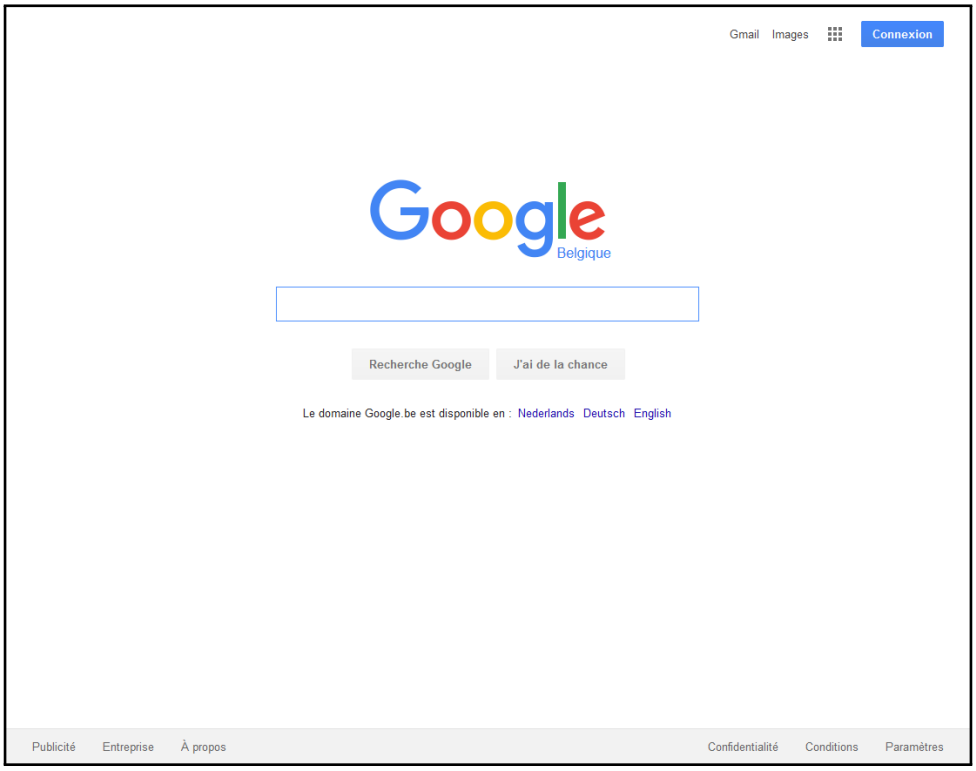


FIGURE 4.29 – Cas Google : page d'accueil

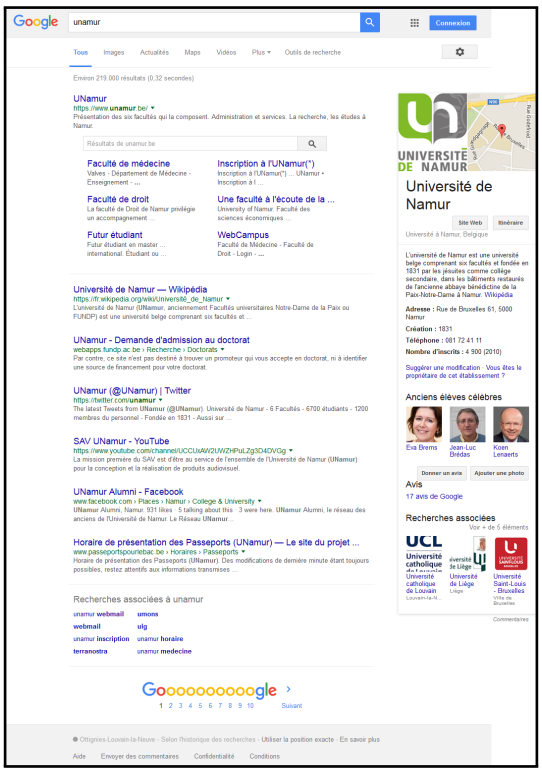


FIGURE 4.30 – Cas Google : résultats de recherche

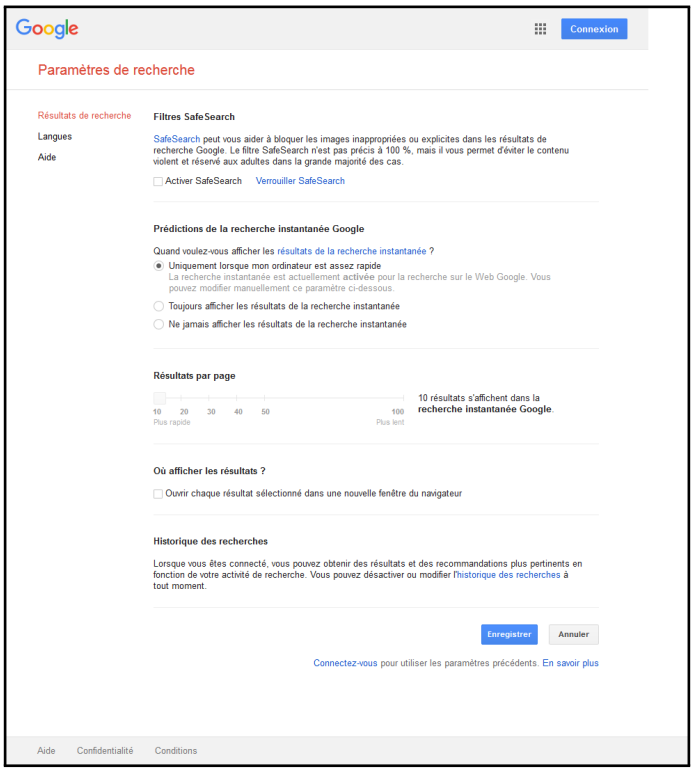


FIGURE 4.31 – Cas Google : paramètres

4.6.2 Modèle d'interface utilisateur

Sur ces trois pages, nous pouvons identifier les types d'éléments (accesseurs) repris ci-dessous. En plus de ceux-ci, nous pourrons modéliser la "Page Web" et le "Navigateur Web" qui contiennent ces éléments.

FIGURE 4.32 – Cas Google : éléments du modèle IU

A partir de ces éléments, nous avons imaginé le modèle de la Figure 4.33 qui illustre ces accesseurs avec des méthodes qui nous permettent d'agir dessus.

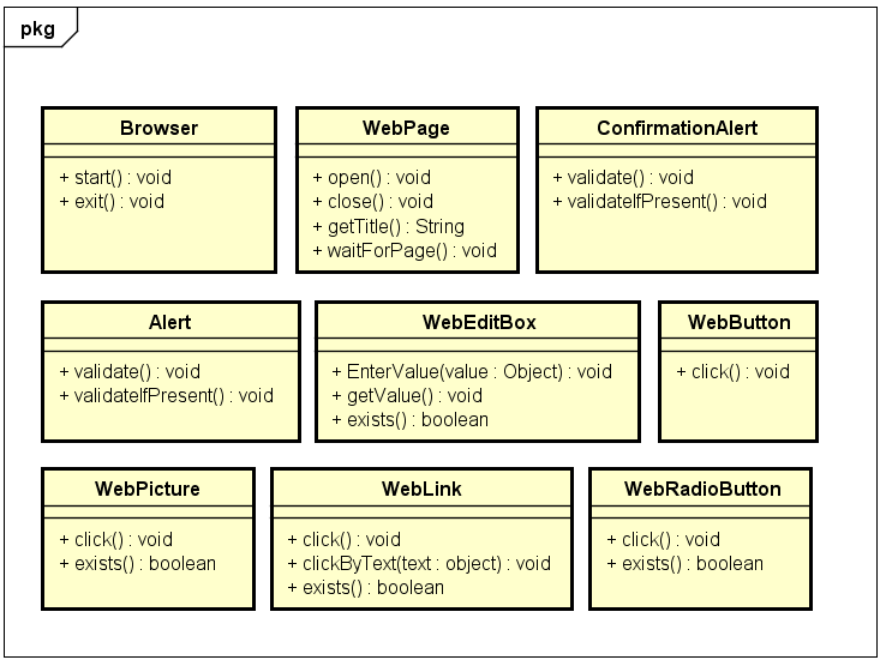


FIGURE 4.33 – Cas Google : classes du modèle IU

Ce modèle IU peut être assez clairement être transcrit en code, comme le montre le fichier Python ci-dessous.

```
# encoding= utf-8

from qtaste import *

#-------------------------------------------
#Browser
class Browser:

    def __init__(self):
        self.api = testAPI.getSelenium(INSTANCE_ID='Google')

    def start(self):
        self.api.openBrowser(testData.getValue("BROWSER"))
        self.api.windowMaximize()

    def exit(self):
        self.api.stop()

#-------------------------------------------
#WebPage
class WebPage:

    def __init__(self, address, timeout):
        self.address = address
        self.timeout = timeout
        self.api = testAPI.getSelenium(INSTANCE_ID='Google')

    def open(self):
        self.api.open(self.address)

    def close(self):
        self.api.close()

    def getTitle(self):
        return self.api.getTitle()

    def waitForPage(self):
        import time
        i = 0
        while i < 5:
            time.sleep(1)
            if self.api.getEval("document.readyState;") == "complete" :
    return
            i += 1

#-------------------------------------------
#Alert
class Alert:

    def __init__(self):
        self.api = testAPI.getSelenium(INSTANCE_ID='Google')

    def validate(self):
        self.api.getAlert()

    def validateIfPresent(self):
        try:
            self.api.getAlert()
        except:
            return
```

```
    def exists(self):
        return self.api.isAlertPresent()

#-----------------------------------------
#ConfirmationAlert
class ConfirmationAlert:

    def __init__(self):
        self.api = testAPI.getSelenium(INSTANCE_ID='Google')

    def validate(self):
        self.api.getConfirmation()

    def validateIfPresent(self):
        try:
            self.api.getConfirmation()
        except:
            return

    def exists(self):
        return self.api.isConfirmationPresent()


#-----------------------------------------
#WebEditBox
class WebEditBox:

    def __init__(self, accessMethod, accessValue):
        self.accessMethod = accessMethod
        self.accessValue = accessValue
        self.api = testAPI.getSelenium(INSTANCE_ID='Google')

    def enterValue(self, value):
        self.api.type(self.accessMethod + "=" + self.accessValue ,
    value )

    def getValue(self):
        value = self.api.getText(self.accessMethod + "=" + self.
    accessValue )
        return value

    def exists(self):
        if (self.api.isElementPresent(self.accessMethod + "=" +  self.
    accessValue)):
            return self.api.isVisible(self.accessMethod + "=" +  self.
    accessValue)
        else:
            return False

#-----------------------------------------
#Webbutton
class WebButton:

    def __init__(self, accessMethod, accessValue):
        self.accessMethod = accessMethod
        self.accessValue = accessValue
        self.api = testAPI.getSelenium(INSTANCE_ID='Google')

    def click(self):
```

```
        if not(self.api.isVisible(self.accessMethod + "=" +  self.
    accessValue)):
            raise Exception('Button not found')
        self.api.clickAt(self.accessMethod + "=" + self.accessValue, "
    0,0" )

    def exists(self):
        if (self.api.isElementPresent(self.accessMethod + "=" +  self.
    accessValue)):
            return self.api.isVisible(self.accessMethod + "=" +  self.
    accessValue)
        else:
            return False


#--------------------------------------------
#Weblink
class WebLink:

    def __init__(self, accessMethod, accessValue):
        self.accessMethod = accessMethod
        self.accessValue = accessValue
        self.api = testAPI.getSelenium(INSTANCE_ID='Google')

    def click(self):
        self.api.click(self.accessMethod + "=" + self.accessValue )


    def clickByText(self, text):
        self.api.click("link=" + text )

    def exists(self):
        if (self.api.isElementPresent(self.accessMethod + "=" +  self.
    accessValue)):
            return self.api.isVisible(self.accessMethod + "=" +  self.
    accessValue)
        else:
            return False



#--------------------------------------------
#WebRadiobutton
class WebRadioButton:

    def __init__(self, accessMethod, accessValue):
        self.accessMethod = accessMethod
        self.accessValue = accessValue
        self.api = testAPI.getSelenium(INSTANCE_ID='Google')

    def click(self):
        self.api.clickAt(self.accessMethod + "=" + self.accessValue, "
    0,0" )

    def exists(self):
        if (self.api.isElementPresent(self.accessMethod + "=" +  self.
    accessValue)):
            return self.api.isVisible(self.accessMethod + "=" +  self.
    accessValue)
        else:
```

```
            return False


#-----------------------------------------
#WebPicture
class WebPicture:

    def __init__(self, accessMethod, accessValue):
        self.accessMethod = accessMethod
        self.accessValue = accessValue
        self.api = testAPI.getSelenium(INSTANCE_ID='Google')

    def click(self):
        self.api.clickAt(self.accessMethod + "=" + self.accessValue, "
    0,0" )

    def exists(self):
        if (self.api.isElementPresent(self.accessMethod + "=" +  self.
    accessValue)):
            return self.api.isVisible(self.accessMethod + "=" +  self.
    accessValue)
        else:
            return False
```

Modèle d'interface utilisateur de Google

4.6.3 *Mapping* de l'interface utilisateur

Nous avons ensuite identifié tous les éléments concrets auxquels nous devions avoir accès (les instances des accesseurs). Le fichier Python suivant a été généré par l'*AddOn* AbsCon.

```
# encoding= utf-8
#Imports
import uimodel_web

#mapping definition
googlePage = uimodel_web.WebPage(
                    "https://www.google.be/",
                    5000)
searchBar = uimodel_web.WebEditBox(
                    "id",
                    "lst-ib")
searchButton = uimodel_web.WebButton(
                    "name",
                    "btnK")
parametersButton = uimodel_web.WebButton(
                    "id",
                    "ab_opt_icon")
parametersLink = uimodel_web.WebLink(
                    "link",
                    "Parametres de recherche")
disableInstantSearchButton = uimodel_web.WebRadioButton(
                    "xpath",
                    "//div[@id='instant-radio']/div[3]/span")
```

```
enableInstantSearchButton = uimodel_web.WebRadioButton(
                        "xpath",
                        "//div[@id='instant-radio']/div[2]/span")
searchResultLink= uimodel_web.WebLink(
                        "link",
                        "")
submitParametersButton = uimodel_web.WebButton(
                    "xpath" ,
                    "//div[@id='form-buttons']/div" )
navPicture        = uimodel_web.WebPicture(
                    "id",
                    "navcnt")
mainLogo          = uimodel_web.WebPicture(
                    "id",
                    "hplogo")
cancelParametersButton        = uimodel_web.WebButton(
                    "xpath",
                    "//div[@id='form-buttons']/div[2]")
confirmation      = uimodel_web.ConfirmationAlert()
alert             = uimodel_web.Alert()
browser           = uimodel_web.Browser()
```

Mapping de l'interface utilisateur de Google généré par AbsCon

4.6.4 Modèle du système

Comme dit précédemment, nous nous bornerons à trois pages, mais nous nous limiterons aussi à une fonctionnalité du moteur de recherche. Dans les paramètres de Google, nous pouvons activer ou désactiver la recherche instantanée. Une fois celle-ci activée, lorsque l'utilisateur entre un mot, Google affiche déjà les résultats sans que l'utilisateur ne clique sur le bouton de recherche. Si la fonction est désactivée, il sera forcé de cliquer sur ce dernier pour voir les résultats.

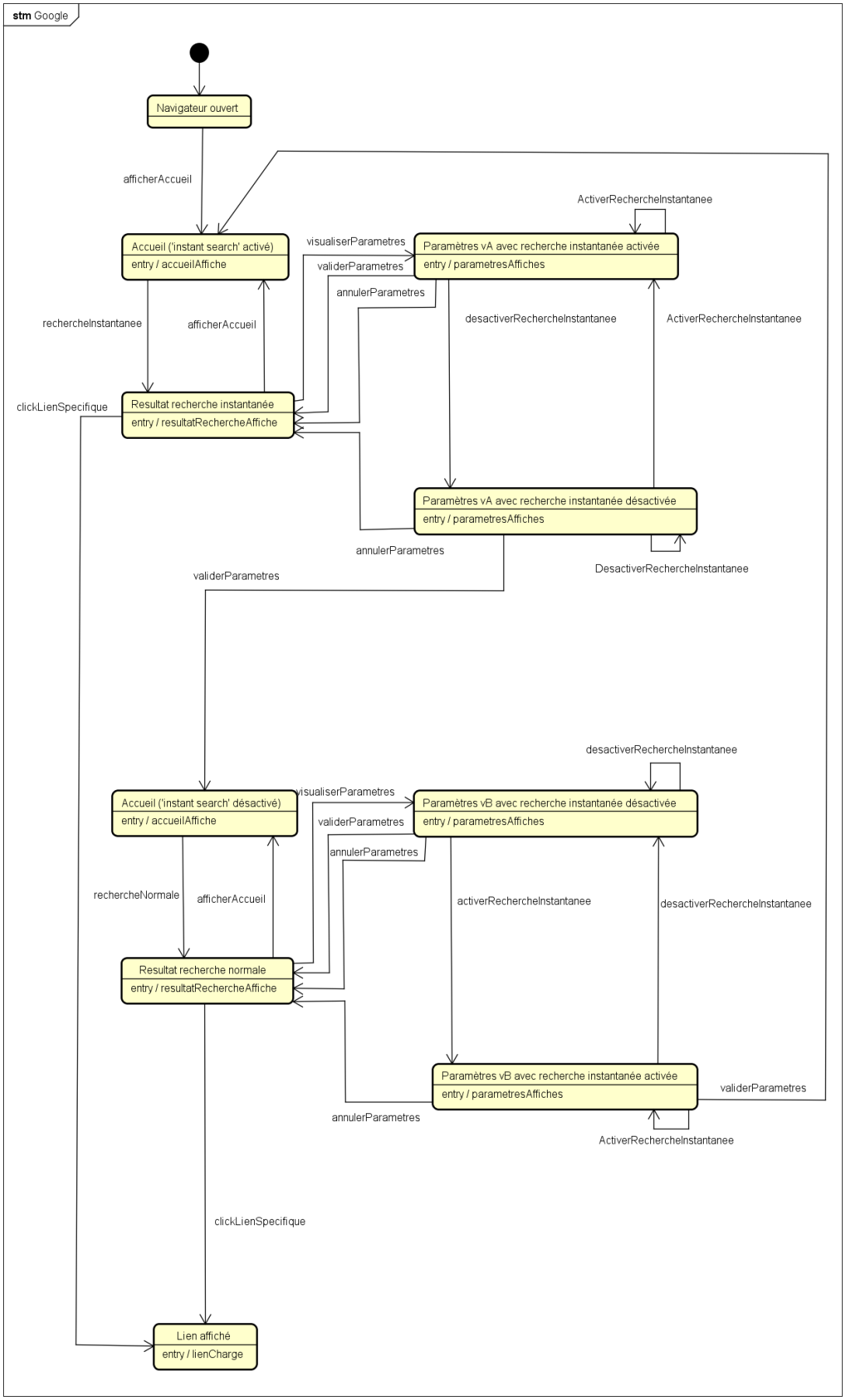


FIGURE 4.34 – Cas Google : modèle du site (limité)

4.6.5 Tests abstraits

Lorsque nous donnons notre modèle à un générateur de tests abstraits tel que VIBeS [16], il en ressort une série de CTAs. Nous en avons sélectionnés quelques-uns qui sont détaillés dans le fichier XML généré par VIBeS :

```
<tests>
   <test>
       <action> start </action>
       <action> afficherAccueil </action>
       <assert> accueilAffiche </assert>
       <action> rechercheInstantanee </action>
       <assert> resultatRechercheAffiche </assert>
       <action> clickLienSpecifique </action>
       <assert> lienCharge </assert>
       <action> exit </action>
   </test>
   <test>
       <action> start </action>
       <action> afficherAccueil </action>
       <assert> accueilAffiche </assert>
       <action> rechercheInstantanee </action>
       <assert> resultatRechercheAffiche </assert>
       <action> afficherAccueil </action>
       <assert> accueilAffiche </assert>
       <action> rechercheInstantanee </action>
       <assert> resultatRechercheAffiche </assert
       <action> visualiserParametres </action>
       <assert> parametresAffiches </assert>
       <action> desactiverRechercheInstantanee </action>
       <assert> parametresAffiches </assert>
       <action> validerParametres </action>
       <assert> resultatRechercheAffiche </assert>
       <action> visualiserParametres </action>
       <assert> parametresAffiches </assert>
       <action> activerRechercheInstantanee </action>
       <assert> parametresAffiches </assert>
       <action> validerParametres </action>
       <assert> resultatRechercheAffiche </assert>
       <action> afficherAccueil </action>
       <assert> accueilAffiche </assert>
       <action> rechercheInstantanee </action>
       <assert> resultatRechercheAffiche </assert>
       <action> exit </action>
   </test>
</tests>
```

CTAs de Google

4.6.6 Définition des opérations

Ensuite, nous avons écrit le code des stimulis et des assertions dans l'*AddOn*. Le fichier suivant a, par la suite, été généré par ce dernier.

```
# encoding= utf-8
#Imports
from qtaste import *
import concretizer_UiMappings

#Actions definition
def start():
    """
    @step       open the browser
    @expected   exit the browser
    """
    concretizer_UiMappings.browser.start()



def afficherAccueil():
    """
    @step       Open the google main page
    @expected   The search bar of google must be available
    """
    concretizer_UiMappings.googlePage.open()



def rechercheInstantanee():
    """
    @step        Enter a value in the search bar of google
    @expected   google searches the value directly
    """

    concretizer_UiMappings.searchBar.enterValue(testData.getValue("
   SEARCHVALUE"))



def clickLienSpecifique():
    """
    @step       click on a research link
    @expected   the link is opened
    """
    concretizer_UiMappings.searchResultLink.clickByText(testData.
   getValue("LINKTOCLICK"))



def exit():
    """
    @step       close the web page
    @expected   the web page is closed
    """
    concretizer_UiMappings.browser.exit()



def visualiserParametres():
    """
    @step        Open the settings section
    @expected   Settings opened
```

```
    """

    #enter in settings section
    concretizer_UiMappings.parametersButton.click()
    concretizer_UiMappings.parametersLink.click()



def desactiverRechercheInstantanee():
    """
    @step       disable instant search
    @expected  The instant search is disabled
    """
    concretizer_UiMappings.disableInstantSearchButton.click()



def annulerParametres():
    """
    @step      cancel parameters
    @expected  parameters are not taken into account
    """
    concretizer_UiMappings.cancelParametersButton.click()

    #agree the alert displayed
    if (concretizer_UiMappings.confirmation.exists()):
        concretizer_UiMappings.confirmation.validate()



def validerParametres():
    """
    @step      validate parameters
    @expected  parameters are taken into account
    """
    concretizer_UiMappings.submitParametersButton.click()

    #agree the alert displayed
    if (concretizer_UiMappings.alert.exists()):
        concretizer_UiMappings.alert.validate()



def rechercheNormale():
    """
    @step       Enter a value in the search bar of google and click the
    search button
    @expected  google searches the value directly
    """

    concretizer_UiMappings.searchBar.enterValue(testData.getValue("
   SEARCHVALUE"))
    concretizer_UiMappings.searchButton.click()



def activerRechercheInstantanee():
    """
    @step       enable instant search
    @expected  The instant search is enabled
```

```
    """
    concretizer_UiMappings.enableInstantSearchButton.click()




#Asserts definition
def accueilAffiche():
    """
    @step       return 1 if the main page is displayed
    @expected  return 1 if the main page is displayed
    """

    concretizer_UiMappings.googlePage.waitForPage()
    return concretizer_UiMappings.mainLogo.exists()



def resultatRechercheAffiche():
    """
    @step       return 1 if the result page is displayed
    @expected  return 1 if the result page is displayed
    """
    import time

    #if the navigation picture is not present, wait 2 seconds and re-
    check
    concretizer_UiMappings.googlePage.waitForPage()
    if (not(concretizer_UiMappings.navPicture.exists())):
        time.sleep(3)
    return concretizer_UiMappings.navPicture.exists()



def lienCharge():
    """
    @step        wait for result page loaded
    @expected   result page is loaded
    """
    concretizer_UiMappings.googlePage.waitForPage()
    return 1



def parametresAffiches():
    """
    @step      check if parameters page is loaded and displayed
    @expected return 1 if true
    """
    import time

    concretizer_UiMappings.googlePage.waitForPage()
    if (concretizer_UiMappings.googlePage.getTitle() == "Parametres de
    recherche"):
        #check if one element of the page is present
        if (not(concretizer_UiMappings.disableInstantSearchButton.
    exists())):
            time.sleep(3)
        return concretizer_UiMappings.disableInstantSearchButton.exists
```

```
    ()

    else:
        return 0
```

Mapping des opérations généré par AbsCon

4.6.7 Définition des données

Pour rester simple dans notre exemple, nous avons défini un unique jeu de données. Il sera représenté par une seule ligne dans notre fichier CSV généré par AbsCon.

```
SEARCHVALUE ; LINKTOCLICK ; BROWSER   ;;
unamur      ; UNamur      ; *firefox  ;;
```

Données de test

4.6.8 Génération des tests concrets

Une fois tous les autres fichiers générés, il ne reste plus qu'à AbsCon de générer le fichier qui appelera les opérations précédemment définies. Voici ce fichier pour le CTA n°2 du point 4.6.5.

```
# encoding= utf-8
#Imports
from qtaste import *
import concretizer_Operations

#Assert
def doAssert(method, message):
    res = method()
    if res == 0:
        raise QTasteTestFailException(message)


#Steps
doStep(concretizer_Operations.start)
doStep(concretizer_Operations.afficherAccueil)
doAssert(concretizer_Operations.accueilAffiche, "assertion
    accueilAffiche has failed")
doStep(concretizer_Operations.rechercheInstantanee)
doAssert(concretizer_Operations.resultatRechercheAffiche, "assertion
    resultatRechercheAffiche has failed")
doStep(concretizer_Operations.afficherAccueil)
doAssert(concretizer_Operations.accueilAffiche, "assertion
    accueilAffiche has failed")
doStep(concretizer_Operations.rechercheInstantanee)
doAssert(concretizer_Operations.resultatRechercheAffiche, "assertion
    resultatRechercheAffiche has failed")
```

```
doStep(concretizer_Operations.visualiserParametres)
doAssert(concretizer_Operations.parametresAffiches, "assertion
    parametresAffiches has failed")
doStep(concretizer_Operations.desactiverRechercheInstantanee)
doAssert(concretizer_Operations.parametresAffiches, "assertion
    parametresAffiches has failed")
doStep(concretizer_Operations.annulerParametres)
doAssert(concretizer_Operations.resultatRechercheAffiche, "assertion
    resultatRechercheAffiche has failed")
doStep(concretizer_Operations.afficherAccueil)
doAssert(concretizer_Operations.accueilAffiche, "assertion
    accueilAffiche has failed")
doStep(concretizer_Operations.rechercheInstantanee)
doAssert(concretizer_Operations.resultatRechercheAffiche, "assertion
    resultatRechercheAffiche has failed")
doStep(concretizer_Operations.clickLienSpecifique)
doAssert(concretizer_Operations.lienCharge, "assertion lienCharge has
    failed")
doStep(concretizer_Operations.exit)
```

Tests concrets générés par AbsCon

4.6.9 Exécution

Une nouvelle série de tests à exécuter apparaît dans l'interface principale de QTaste. Pour chacun des tests, nous pouvons voir ce qu'il contient et quelles données il va utiliser. On peut ensuite les exécuter un par un ou de manière simultanée. Ci-dessous, quatre captures d'écran du logiciel lors de l'exécution des tests de la figure 4.6.5 concrétisés grâce à l'*AddOn* AbsCon.

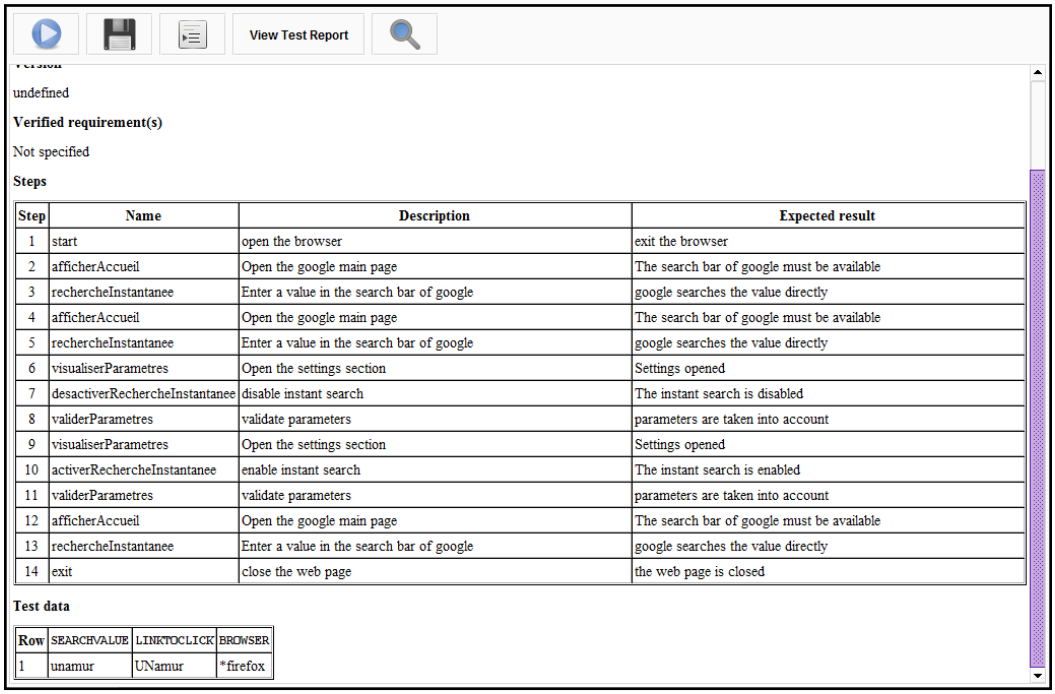


FIGURE 4.35 – Cas Google : visualisation du test dans QTaste

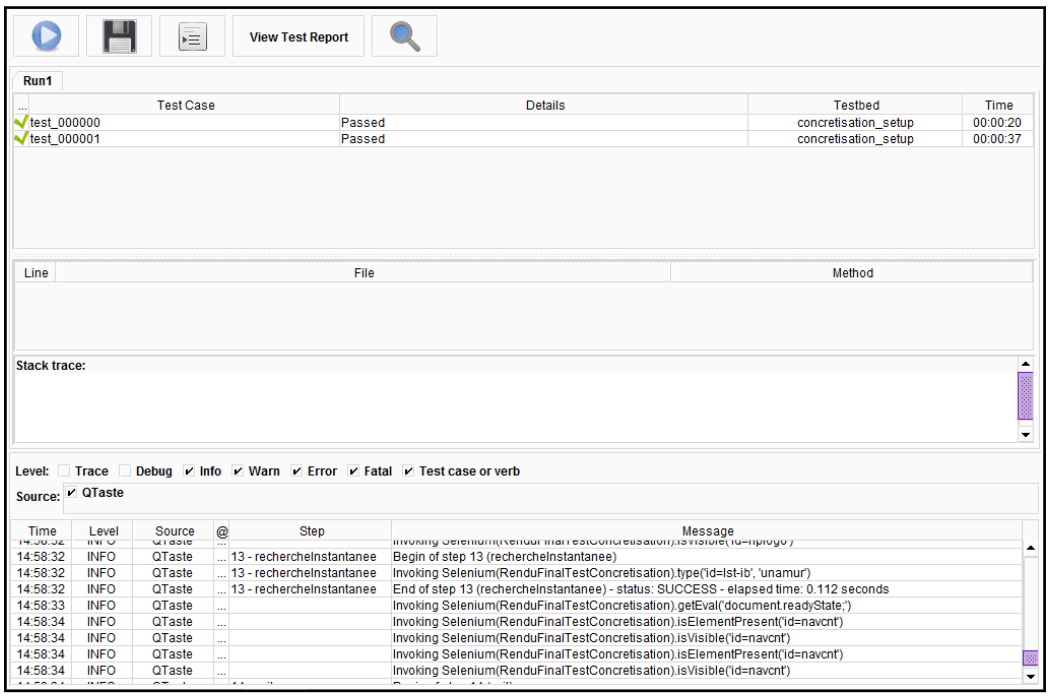
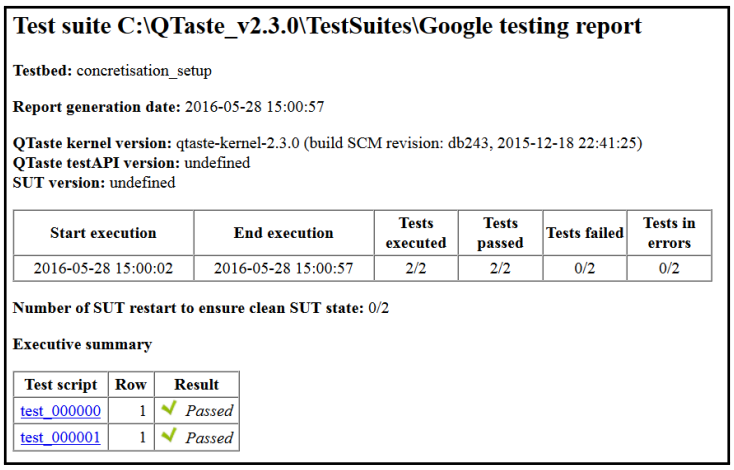


FIGURE 4.36 – Cas Google : exécution des tests dans QTaste



Test suite C:\QTaste_v2.3.0\TestSuites\Google testing report

Testbed: concretisation_setup

Report generation date: 2016-05-28 15:00:57

QTaste kernel version: qtaste-kernel-2.3.0 (build SCM revision: db243, 2015-12-18 22:41:25)
QTaste testAPI version: undefined
SUT version: undefined

Start execution	End execution	Tests executed	Tests passed	Tests failed	Tests in errors
2016-05-28 15:00:02	2016-05-28 15:00:57	2/2	2/2	0/2	0/2

Number of SUT restart to ensure clean SUT state: 0/2

Executive summary

Test script	Row	Result
test_000000	1	*Passed*
test_000001	1	*Passed*

FIGURE 4.37 – Cas Google : en-tête du rapport de tests

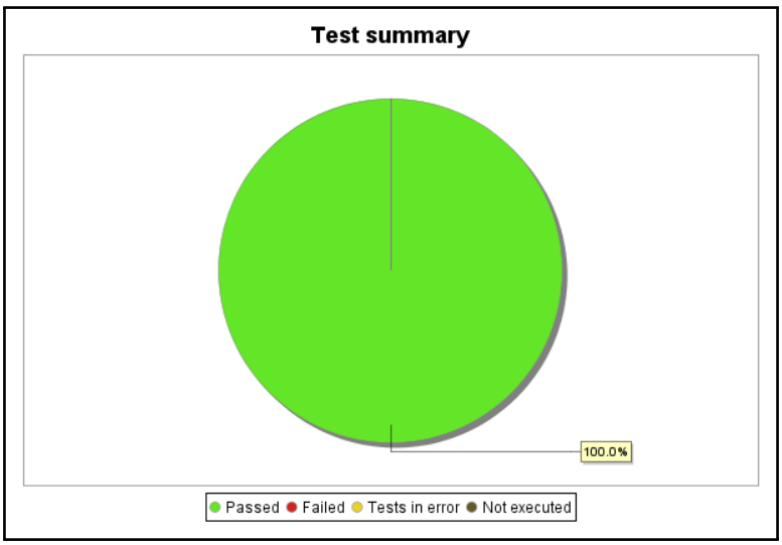


FIGURE 4.38 – Cas Google : graphique du rapport de tests

Test script test_000000

Row	Status	Description	Data
1 -	Passed	Passed	

SEARCHVALUE unamur
LINKTOCLICK UNamur
BROWSER *firefox

Step	Step description	Expected result	
1	open the browser	exit the browser	Passed
2	Open the google main page	The search bar of google must be available	Passed
3	Enter a value in the search bar of google	google searches the value directly	Passed
4	click on a research link	the link is opened	Passed
5	close the web page	the web page is closed	Passed

FIGURE 4.39 – Cas Google : détails des opérations dans le rapport de tests

Chapitre 5

Evaluation

Dans ce chapitre, nous allons évaluer AbsCon, le fruit de ce travail. Nous allons définir si les fonctionnalités et les résultats fournis par l'*AddOn* sont réellement novateurs et apportent un réel gain pour la communauté de test.

5.1 Limites du *mapping* de l'interface utilisateur

L'exemple de *mapping* de la section 4.6 était assez court et facile à réaliser ; faire un *mapping* exhaustif pour une application complète peut par contre, demander beaucoup de temps. Un des problèmes qui se pose de manière récurrente est "comment identifier l'élément auquel je souhaite accéder ?". Si le modèle IU est censé fournir des propositions pour identifier les éléments, il faut tout de même une action manuelle de l'utilisateur pour les définir. Dans l'exemple cité de la section 4.6, nous avons dû identifier chaque élément nécessaire dans les trois pages web manuellement. Des navigateurs tels que *Firefox* [3] permettent de voir directement le code source lié à un composant de la page. Cela nous permet donc de voir si l'élément possède un attribut HTML *id* ou *name*. Dans le cas où ce type d'attribut n'est pas disponible, il est toujours possible de déduire le noeud *Xpath* de l'élément. Si à nouveau, ce genre d'opération peut sembler complexe à réaliser, des *AddOns* Firefox permettent de nous assister pour cela [15]. Nous n'avons pas trouvé d'outil qui permettent d'extraire automatiquement les éléments dont nous pourrions avoir besoin sur une page web. Si le besoin s'en faisait toutefois ressentir, il serait sans doute possible de créer une extension *Firefox* pour cela.

De tels outils pour extraire les informations d'identification des IHM d'un système existent aussi pour des applications Windows et Java par exemple. Ils fonctionnent en général selon la même méthodologie : on sélectionne un élément de l'IHM et il nous affiche le nom, l'ID, le texte, etc... Pour Windows, un inspecteur d'IHM est fourni dans le SDK de Microsoft [6]. Pour Java, une multitude d'outils existent, dont *Swing Inspector* [12].

Le problème du *mapping* IU reste tout de même contraignant pour les grosses applications ; mais il s'agit d'un problème séparé. Une solution indépendante qui ferait une extraction automatique des éléments d'une IHM pourrait être développée par la suite.

5.2 Temps d'exécution

Nous partirons du postulat que les CTAs sont déjà générés pour cette étape. Ensuite, nous pourrons comparer l'exécution manuelle ou automatique de ces derniers.Pour les automatiser, nous devons concrétiser ces CTAs. Pour cela, nous devrons faire face à deux opérations chronophages.
Tout d'abord il y a la rédaction du modèle IU. Ce point est un peu particulier car lorsqu'un modèle (par exemple, le modèle IU utilisé pour tester les applications web) est écrit, il n'est plus nécessaire d'y toucher si l'on veut tester d'autres applications web. On peut donc imaginer de créer une multitude de modèles IU fournis avec AbsCon de manière à retirer cette tâche de l'équation.
Ensuite, il y a la définition du *mapping* IU ; nous en avons déjà parlé dans le point précédent. Pour un utilisateur rodé, utilisant les fonctions avancées de son navigateur web, l'exemple de la section 4.6 peut être réalisé en une dizaine de minutes.

Après ces deux opérations gourmandes en temps, il reste deux actions à effectuer. D'abord la définition des actions et des assertions. Avec un bon *mapping* IU et un bon modèle IU, il s'agit d'une étape triviale. La durée de cette étape dépendra uniquement du nombre d'opérations abstraites à définir. Enfin, il reste la définition des données de tests. Ici, le temps dépend du nombre de données que l'on souhaite tester.
Pour un type connu d'application (ayant la même envergure que notre exemple), avec des CTAs déjà générés, on peut retenir qu'il faut moins d'une heure pour générer des tests automatisés. Ensuite, chaque test s'exécutera à une vitesse légèrement supérieure à celle d'une exécution manuelle par un expert de l'application.

Nous souhaitons comparer cet ordre de temps avec une exécution manuelle des CTAs, nous partirons du même postulat : le type d'application est déjà maîtrisé. Ensuite, le seul temps que nous devons prendre en compte est le temps d'exécution de la personne dédiée.

Sur une série de deux tests comme le propose l'exemple de la section 4.6, l'intérêt de AbsCon est minime. Par contre, sur une série de plusieurs milliers de tests, on peut apercevoir un intérêt certain. Tout d'abord, il n'y a pas le facteur d'erreur humaine. Lorsqu'un test est réussi, nous avons des logs pour le vérifier. Ensuite, avec un nombre si élevé de tests, nous pourrions voir une réelle amélioration du temps d'exécution. En effet, AbsCon ne se fatigue pas à l'inverse du testeur manuel. Enfin, pas besoin de ressources humaines pour exécuter les tests, ce qui veut dire que nous pouvons les exécuter jour et nuit sans interruptions.

Nous pouvons donc retenir comme conclusion que pour jouer des tests récurrents, AbsCon est une excellente réponse par rapport à l'exécution manuelle des tests.

5.3 Modificabilité

Lorsque nous avons défini notre architecture avec la Figure 3.7, nous l'avions défendue en argumentant qu'elle offrait une grande modificabilité. Nous allons vérifier cette affirmation ici. Imaginons deux cas de figure :

— Nous avons des tests automatisés réalisés via AbsCon.

— Nous avons écrit des tests QTaste sans modèle IU et sans *mapping* IU. Tout le code est contenu dans le fichier appelé par QTaste.

Ensuite, imaginons que depuis l'écriture des tests l'IHM de Google a légèrement changée. Tout d'abord, l'ID du bouton avec la roue dentée a changé et ensuite, le texte du lien qui permet d'accéder à la page des paramètres a lui aussi changé :

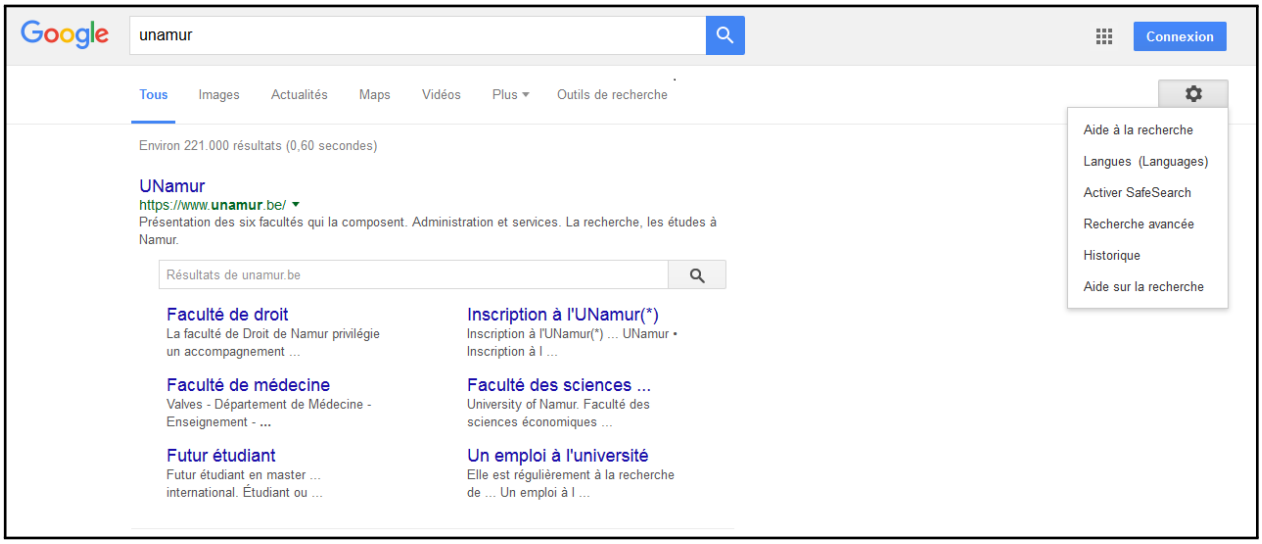


FIGURE 5.1 – L'IHM de Google a été modifiée

Maintenant, reprenons nos deux cas de figure. Dans le premier cas, il suffira de modifier deux *mappings*. Dans le deuxième cas, il faudra vérifier dans tous les fichiers (potentiellement des milliers) et modifier des valeurs là où elles sont utilisées.

Imaginons maintenant que Google modifie à nouveau son IHM et que le fait d'appuyer sur la roue dentée nous permette d'accéder directement au paramètre de la recherche instantanée. Dans le premier cas de figure, nous devrons uniquement modifier l'action "visualierParametres", tandis que, dans le deuxième cas de figure, nous devrons à nouveau parcourir tous les fichiers.

Poussons le concept un peu plus loin et supposons que nous voulons tester ces trois écrans de Google en français et en anglais. Dans le deuxième cas de figure, nous devrons dupliquer tous les tests et modifier la valeur des textes recherchés dans l'IHM. Dans le premier cas de figure, nous n'aurons qu'à utiliser les valeurs contenues dans les données pour trouver les éléments de l'IHM. Prenons l'exemple de notre lien. Le *mapping* AbsCon ressemble initialement à ceci :

```
parametersLink = web.WebLink[
 accessMethod : "link",
 accessValue :"Parametres de recherche"
]
```

Mapping IU

Plutôt que d'écrire la valeur du texte dans le *mapping*, ajoutons-la dans les données :

```
SEARCHVALUE ; LINKTOCLICK ; BROWSER  ; PARAMETERLINKTEXT          ;;
unamur      ; UNamur      ; *firefox ; Parametres de recherche ;;
unamur      ; UNamur      ; *firefox ; Research settings        ;;
```

Données de test

Ensuite, il suffit de redéfinir le *mapping* comme ceci :

```
parametersLink = web.WebLink[
 accessMethod : "link",
 accessValue : testData.getValue("PARAMETERLINKTEXT")
]
```

Mapping IU

Nous l'avons donc montré ci-dessus, l'architecture choisie est non seulement simple mais aussi très efficace.

5.4 Discussion de la validité

Pour finaliser ce document, nous avons dû prendre certaines décisions et effectuer certains choix. Dans ce chapitre, nous allons discuter de la validité de ces choix.

Le premier choix réalisé a été d'abandonner très vite les solutions existantes pour en proposer une plus personnelle. Cette décision peut en effet être discutable mais elle se justifie par le fait que les documents scientifiques se basaient trop souvent sur des solutions incomplètes et que les solutions commerciales, elles, étaient non libres de droit et bien souvent, payantes.

Ensuite, nous avons choisi QTaste [7] comme plateforme d'exécution de tests automatisés. Cette décision peut sembler audacieuse lorsque l'on sait que c'est une plateforme maintenue par une petite société dont ce n'est pas l'activité principale. De plus, il n'existe aucune communauté sur le web qui permet de fournir du support, alors que la documentation est relativement limitée. A côté de cela, QTaste reste néanmoins un support excellent grâce à son aspect "universel", et son implémentation *open source* peut offrir une réponse acceptable pour le manque de communauté qui aurait pu apporter un quelconque support. AbsCon pourrait être le point de départ pour créer un forum d'aide en ligne, histoire de lancer le mouvement pour QTaste.

Nous avons aussi choisi d'exclure complètement l'utilisation de STALE [22] alors que théoriquement, il pouvait concrétiser des tests en Python destinés à s'exécuter via QTaste. Cependant, son utilisation est assez compliquée alors que sa syntaxte est censée produire l'effet inverse. De plus, STALE utilise un algorithme qui, lorsqu'on veut concrétiser des tests destinés à QTaste, est inutile. AbsCon a prouvé qu'avec une bonne méthodologie on pouvait produire quelque chose d'équivalent en terme de quantité de code à fournir (pas de réplication inutile de code). En outre, l'utilisation de STALE demandait un outil supplémentaire, alors que AbsCon, lui, est intégré à QTaste. Par contre, STALE, propose un *constraint solver* qui permet la génération de données de tests aléatoires et contrôlées. Si un tel *constraint solver* n'a pas encore été implémenté dans AbsCon, son architecture le permettrait sans aucune difficulté.

En ce qui concerne précisément cette méthodologie, on pourrait se demander si elle est réellement inédite puisqu'elle propose, finalement, de coder correctement en utilisant des librairies et des fonctions. En réalité, elle va plus loin puisqu'elle propose de décrire l'IHM des logiciels (ou autres types d'interfaces) à tester via un diagramme de classe et ensuite d'instancier tous les accesseurs via les constructeurs des classes et enfin, une fois que les opérations sont décrites, de pouvoir exécuter les tests sur n'importe quel type de système grâce à QTaste.

AbsCon est une implémentation de cette méthodologie. Son but, dans ce document, est de démontrer que la méthode fonctionne. C'est pourquoi nous avons implémenté toutes les fonctions nécessaires, mais pas toujours celles qui ont été considérées comme des *must have*. C'est la raison pour laquelle, sur l'écran de la figure Figure 4.24 on ne peut, par exemple, pas éditer de manière très agréable les *mappings* déjà définis.

Finalement, dans notre cas d'étude, nous avons fait beaucoup d'estimations temporelles sans pour autant apporter de chiffres exacts. C'est malheureusement lié au fait qu'une personne ne code pas aussi vite qu'une autre. Le même principe s'applique pour l'exécution de tests manuels. C'est pour cela que nous n'avons pas mis en évidence la rapidité d'exécution des tests puisque nous ne pouvions pas la prouver directement. Cela nécessiterait la mise en place d'une *controlled experiment*, et ceci dépasse le cadre de ce travail. Par contre, dans la section suivante, nous détaillerons le protocole de cette potentielle *controlled experiment*.

5.5 Protocole de la *controlled experiment*

Si réaliser une expérience pour tester l'efficacité d'AbsCon sort du cadre de ce travail, nous pouvons néanmoins détailler comment elle devrait se dérouler.

Tout d'abord, il faut mettre en évidence la question à laquelle nous souhaitons trouver une réponse à l'issue de cette expérience : "Dans quelle mesure AbsCon permet-il ou non d'économiser du temps dans le cadre d'une exécution de tests basée sur des tests abstraits ?". Pour trouver une réponse à cette question, nous aurons besoin de répondre à trois sous-questions :

— "Quelle est la population visée ?" : nous viserons une population de testeurs ayant des connaissances de base en programmation.

— "Comment répondre à notre question initiale ?" : une fois l'expérience finie, nous analyserons les données mesurées.

— "Comment mesurer les résultats de l'expérience ?" : nous comparerons les temps d'exécution automatiques par rapport aux exécutions manuelles.

Pour cette expérience, nous aurions besoin de candidats testeurs. Un minimum de quatre personnes serait conseillé. Chaque testeur devra tester les deux méthodes de cette expérience, à savoir, AbsCon et l'exécution manuelle. Chaque candidat devra avoir un niveau intermédiaire en Python et au minimum, un niveau débutant en Java (de manière à pouvoir interpréter les modèles IU).

En ce qui concerne le recrutement de ces candidats, il pourrait se faire auprès de personnes qui s'intéresseraient à l'automatisation de tests (que ce soit sur internet ou via du bouche à oreille). Il pourrait aussi être intéressant de contacter certaines compagnies *IT* qui testent manuellement leurs logiciels. Nous aurions ainsi un panel de testeurs aux manières de fonctionner différentes.

L'expérience en elle-même se déroulerait comme suit :

— Nous commencerions par énoncer les instructions aux participants. Ils auront tous une application web à tester. Il leur sera fourni un modèle de l'application ainsi que des CTAs au format graphique et XML (pour AbsCon). Il faudra un nombre assez conséquent de CTAs (>50) ; sinon il n'y aurait aucun intérêt à les automatiser. Il leur sera aussi fourni un modèle d'interface utilisateur pour applications/sites web destiné à être utilisé dans AbsCon. Ils seront d'abord chronométrés pour exécuter tous les tests manuels sur le système. Ensuite, ils seront chronométrés pour concrétiser les tests avec AbsCon. Un deuxième temps sera enregistré lors de l'exécution automatique de ces tests.

— Des essais pratiques devront être réalisés avec les participants pour qu'ils puissent utiliser AbsCon et qu'ils comprennent bien l'application web à tester.

— Déroulement de l'expérience.

— Une fois l'expérience terminée, il faudra soumettre un questionnaire à chaque participant suivi d'une interview. Il faudra déterminer si le testeur a été limité ou dérangé par certaines fonctionnalités d'AbsCon. Il pourra aussi proposer des améliorations à apporter.

— Une fois que tous les testeurs ont fini l'expérience, il est temps de débriefer et de mettre en forme toutes les mesures obtenues.

Avec ces mesures, nous pourrions obtenir trois temps moyens. Tout d'abord, le temps moyen d'exécution des tests manuels ($Tmanu$). Ensuite, nous pourrons aussi en retirer le temps moyen nécessaire pour concrétiser les CTAs via AbsCon ($Tconc$). Enfin, le temps moyen d'exécution automatique via QTaste ($Tauto$).

Le temps de concrétisation des CTAs ($Tconc$) devra être considéré comme un temps unique, alors que les temps d'exécutions ($Tmanu$ et $Tauto$) doivent être interprétés comme des temps nécessaires à chaque fois que l'on veut tester le système.

Pour mesurer l'efficacité d'AbsCon sur le système testé, il faut déterminer combien de répétitions d'exécution de tests sont nécessaires pour le rentabiliser. Nous pourrons le faire via l'équation suivante, où x est le nombre de répétitions minimal à déterminer :

$$x * Tmanu = Tconc + x * Tauto$$

Il suffira d'arrondir x à l'entier supérieur pour obtenir le nombre de répétition(s) nécessaire(s) pour rentabiliser le processus de concrétisation.

Chapitre 6

Conclusion

L'idée de cette étude est née grâce à VIBeS [16]. En effet, cet outil est en mesure de créer de grands nombres de CTAs sur base d'un modèle mais il ne proposait aucune solution pour les exécuter de manière automatique. Lorsque nous avions analysé le problème, nous nous sommes demandé s'il était possible de concrétiser et d'exécuter des tests en suivant une méthode commune, quel que soit le système concerné.

Nous avons donc décomposé le problème en deux sous-problèmes : la concrétisation et l'exécution. Nous avons tenté de trouver des outils existants pour réaliser ces deux tâches tout en gardant à l'esprit que VIBeS devait pouvoir s'interfacer avec le concrétisateur (ou l'inverse). Lors de cette analyse de l'existant, nous nous sommes donné comme consigne d'utiliser des produits *open source* de manière à pouvoir proposer une chaîne complète qui le soit elle aussi.

De cette recherche ont découlé deux choix différents résultant l'un de l'autre. Tout d'abord, nous avons décidé d'utiliser QTaste [7] pour l'exécution des tests. Il s'agit d'une plateforme qui offre une architecture inédite. Grâce à elle, on peut prétendre pouvoir tester tout type de système (aucun contre-exemple théorique n'a été trouvé). Ensuite, en sachant quel environnement de test serait utilisé, nous avons pris le parti de développer notre propre méthodologie de concrétisation et de l'intégrer à l'outil QTaste via l'*AddOn* AbsCon.

Dans notre cas d'étude, nous avons pu démontrer que la solution fonctionne comme attendu. Si elle demande plus de temps en terme de préparation que les tests manuels (modèle IU, *mapping* IU, *mapping* des opérations, etc...), son intérêt est évident lorsqu'il est question de manipuler des centaines/milliers (voire plus) de tests et que l'on pourrait être amené à les rejouer plusieurs fois.

Si la proposition faite via AbsCon convainc les utilisateurs, ses perspectives sont grandes. En effet, ses sources sont disponibles sur GitHub [29] et il sera proposé aux utilisateurs de partager tout nouveau modèle IU/ *Test API* s'appliquant à un type de système pouvant être utilisé par un grand nombre.

A côté de cela, il est envisagé de faire évoluer l'outil dans un futur proche. Tout d'abord, nous souhaitons mettre en place un *constraint solver* offrant, au minimum, les mêmes possibilités que celui de STALE [22]. Ensuite, une meilleure gestion des erreurs ainsi que de leur affichage ; qui se fait dans la console actuellement. Enfin,

améliorer l'écran de définition du *mapping* de l'interface utilisateur, pour permettre une édition plus aisée des *mappings* définis.

Pour revenir à la question initiale, nous pouvons répondre que oui, des solutions à ces deux problèmes existaient. Par contre ces outils ne nous ont pas satisfaits en raison du fait qu'ils sont non-libres ou parce que les solutions qu'ils proposent ne sont pas efficaces. Nous avons donc fourni une solution qui satisfait à tous nos critères et qui nous a personnellement convaincu en terme d'efficacité.

Annexes

AbsCon sur GitHub

L'implémentation de l'*AddOn* AbsCon est disponible sur GitHub [29]. Le projet est *open source* sous une licence publique générale GNU. Toutes les informations nécessaires pour sa compilation et son intégration dans QTaste sont fournies sur le répertoire GitHub. Il s'agit d'un projet Java dont l'envergure pourrait être évaluée grâce aux données suivantes :

— 34 classes

— 248 méthodes

— 4864 lignes de code

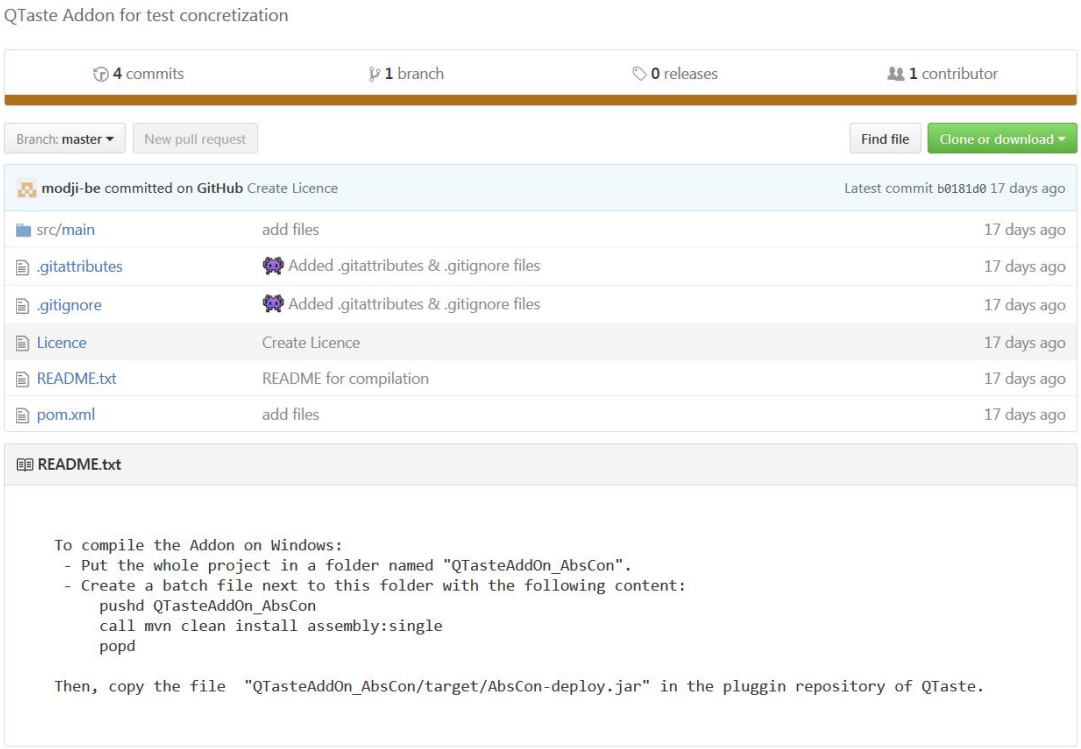


FIGURE 6.1 – Le projet AbsCon sur GitHub

Bibliographie

[1] <https://wiki.python.org/jython/JythonFaq>. [Online ; accessed 19-april-2016].

[2] "autohotkey automation hotkeys scripting". <https://autohotkey.com/>. [Online ; accessed 30-march-2016].

[3] "firefox". <https://www.mozilla.org/fr/firefox/products/>. [Online ; accessed 28-may-2016].

[4] "github qspin/qtaste". <https://github.com/qspin/qtaste/tree/addon>. [Online ; accessed 19-may-2016].

[5] "google". <https://google.be>. [Online ; accessed 26-may-2016].

[6] "inspect.exe". <https://msdn.microsoft.com/en-us/library/windows/desktop/dd318521%28v=vs.85%29.aspx/>. [Online ; accessed 28-may-2016].

[7] "qspin tailored automated system test environment". <https://github.com/qspin/qtaste>. [Online ; accessed 10-april-2016].

[8] "sahi introduction". <http://sahipro.com/docs/introduction/index.html>. [Online ; accessed 30-march-2016].

[9] "seleniumhq browser automation". <http://www.seleniumhq.org/>. [Online ; accessed 30-march-2016].

[10] "sikuli script". <http://www.sikuli.org/>. [Online ; accessed 30-march-2016].

[11] "squish gui tester". <http://www.froglogic.com/squish/gui-testing/>. [Online ; accessed 30-march-2016].

[12] "swing inspector". <http://www.swinginspector.com/index_en.htm>. [Online ; accessed 28-may-2016].

[13] "test api selenium for qtaste". <https://github.com/qspin/qtaste/blob/master/demo/testapi/src/main/java/com/qspin/qtaste/testapi/api/Selenium.java>. [Online ; accessed 26-may-2016].

[14] "what is technical review in software testing ?". <http://istqbexamcertification.com/what-is-technical-review-in-software-testing>. [Online ; accessed 30-march-2016].

[15] "xpatch checker extension website". <https://code.google.com/archive/p/xpathchecker/>. [Online ; accessed 28-may-2016].

[16] "vibes : Variability intensive behavioural testing". <https://projects.info.unamur.be/vibes/index.html>, January 2015. [Online ; accessed 10-april-2016].

[17] Larry Apfelbaum and John Doyle. "model-based testing". *Software Quality Week Conference*, pages 296–300, May 1997.

[18] Robert V. Binder, Anne Kramer, and Bruno Legeard. "2014 model-based testing user survey : Results". <http://model-based-testing.info/wordpress/wp-content/uploads/2014_MBT_User_Survey_Results.pdf>. [Online ; accessed 30-march-2016].

[19] Kalou Cabrera Castillos, Frédéric Dadeau, and Jacques Jullian. "coverage criteria for model-based testing using property patterns". *Ninth Workshop on Model-Based Testing (MBT 2014)*, pages 29–43, 2014.

[20] James S. Collofello. "introduction to software verification and validation". *SEI Curriculum Module SEI-CM-13-1.1*, December 1988.

[21] Jonathan Lasalle, Fabien Peureux, and Jérôme Guillet. "automatic test concretization to supply end-to-end mbt for automotive mechatronic systems". *ETSE '11 Proceedings of the First International Workshop on End-to-End Test Script Engineering*, pages 16–23, July 2011.

[22] Nan Li and Jeff Offutt. "a test automation language framework for behavioral models", April 2015.

[23] Aditya P. Mathur. "foundations of software testing". 2008.

[24] Nicolas Sannier. "modeling requirements requirements verification and validation". <https://nicolassannier.files.wordpress.com/2011/04/3-modeling-requirements-requirements-validation-and-verification-sannier.pdf>. [Online ; accessed 30-march-2016].

[25] Gerardo Schneider. "model-based testing - selecting your tests". <http://www.cse.chalmers.se/edu/year/2012/course/DIT848/files/06-Selecting-Tests.pdf>. [Online ; accessed 30-march-2016].

[26] Sami Taktak. "test et validation du logiciel". <http://cedric.cnam.fr/~taktaks/GLG101/teststructurel.pdf>. [Online ; accessed 17-april-2016].

[27] René Christian Tuyishime. "how to generate selenium test cases with matelo ?". <http://www.all4tec.net/doc_download/243-app-note-how-to-generate-selenium-test-cases-with-matelo>. [Online ; accessed 24-may-2016].

[28] Mark Utting and Bruno Legeard. "practical model-based testing : A tools approach". 2007.

[29] Jeremy Vanhecke. "github modji-be/abscon". <https://github.com/modji-be/AbsCon/>, August 2016. [Online ; accessed 01-august-2016].